Hardcore COMPUTIST'S

Book Of Softkeys

Volume II

SoftKey Publishing

THE

BOOK OF SOFTKEYS

volume II

SoftKey Publishing
PO Box 110846-BK
Tacoma, WA 98411

Table of Contents

Introduction

Welcome to the **Book Of Softkeys** Volume II, a compilation of articles from Hardcore COMPUTIST magazine (issues 6 through 10) that explain **how to remove copy-protection** from commercially sold, locked-up and uncopyable software for the Apple][series of computer systems.

Hardcore COMPUTIST (now just called **COMPUTIST**) is a monthly publication devoted to the serious user of Apple][computers and compatibles. Our magazine contains information you are not likely to find in any of the other major journals dedicated to the Apple market.

Our editorial policy is that we do NOT condone software piracy, but we do believe that honest users are entitled to backup commercial disks they have purchased. In addition to the security of a backup disk, the removal of copy-protection gives the user the option of modifying application programs to meet his or her needs.

Furthermore, the copyright laws guarantee your right to such a deprotected backup copy:

> **..."It is not an infringement for the owner of e copy of e computer program to make or euthorize the making of another copy or adaption of thet computer program provided:**
>
> **1) that such e new copy or adaption is created as an essential step in the utilization of the computer program in conjunction with a machine and that it is used in no other manner, or**
>
> **2) that such new copy or adaption is for archival purposes only and that all archival copies are destroyed in the event that continued possession of the computer program should cease to be rightful.**
>
> **Any exact copies prepared in accordance with the provisions of this section may be leesed, sold, or otherwise transferred, along with the copy from which such copies were prepared, only as part of the lease, sale, or other transfer of all rights in the program. Adeptions so prepared mey be transferred only with the euthorization of the copyright owner."**
>
> United States Code title 17, §117 (17 USC 117)

Those of you who are not already familiar with Hardcore COMPUTIST are advised to read the **What is a Softkey?** article in order to avoid frustration when attempting to follow a softkey or when entering the programs printed in this Book Of Softkeys.

What Is A Softkey?

***Softkey* is a term which we coined to describe a procedure that removes, or at least circumvents, any copy-protection on a particular disk. Once a softkey procedure has been performed, the resulting disk can usually be copied by the use of Apple's COPYA program (on the DOS 3.3 System Master Disk) and is said to be "COPYAable."**

Commands And Controls:

Commands which a reader is required to perform are set apart from normal text by being indented and bold. An example is:

PR#6

The RETURN key must be pressed at the end of every such command unless otherwise specified. Control characters are shown as a single symbol. For example:

6 CTRL P

To complete this command, you must first type the number **6** and then hold the CTRL key while you press the P key.

Requirements:

Most of the programs and softkeys which appear in this book require one of the Apple II series of computers and at least one disk drive with *DOS 3.3*. Occasionally, some programs and procedures have special requirements. The prerequisites for deprotection techniques or programs will always be listed at the beginning of the article under the '**Requirements**:' heading.

Software Recommendations:

The following programs are strongly recommended for readers who wish to obtain the most benefit from our articles:

An Applesoft Program Editor such as *Global Program Line Editor* (GPLE).

A Sector Editor such as *DiskEdit* (SoftKey Publishing), or *ZAP* from *Bag of Tricks* or *Tricky Dick* from *The CIA*.

A Disk Search Utility such as *The Inspector*, *The Tracer* from *The CIA* or *The CORE Disk Searcher* (SoftKey Publishing).

An Assembler such as the *S-C Assembler* or *Merlin/Big Mac*.

A Bit Copier such as *Copy II Plus*, *Locksmith* or *The Essential Data Duplicator*.

A Text Editor capable of producing normal sequential text files such as *Applewriter II*, *Magic Window II* or *Screenwriter II*.

You will also find *COPYA*, *FID* and *MUFFIN* from the *DOS 3.3 System Master Disk* useful.

Super IOB and controllers

Several softkey procedures will make use of a Super IOB controller, a small program that must be keyed into the middle of Super IOB. The controller changes Super IOB so that it can copy different disks. See the **Super IOB 1.5** article and program in this volume of the Book Of Softkeys.

RESET Into The Monitor

Some softkey procedures require that the user be able to enter the Apple's System Monitor (henceforth called the Monitor) during the execution of a copy-protected program. Check the following list to see what hardware you will need to obtain this ability.

Apple][Plus - Apple //e - Apple compatibles:

1) Place an Integer BASIC ROM card in one of the Apple slots.
2) Use a non-maskable interrupt (NMI) card such as *Replay* or *Wildcard.*

Apple][Plus - Apple compatibles:

Install an F8 ROM with a modified RESET vector on the computer's motherboard as detailed in the **'Modified ROMs'** article of Hardcore COMPUTIST # 6.

Apple //e - Apple //c:

Install a modified CD ROM on the computer's motherboard. Clay Harrell's company (Cutting Edge Ent.; Box 43234 Ren Cen Station-HC; Detroit, MI 48243) sells a hardware device that will give you this ability. Making this modification to an Apple //c will void its warranty but the increased ability to remove copy-protection may justify it.

Recommended Literature

Apple][Reference Manual
DOS 3.3 manual
Beneath Apple DOS
by Don Worth and Pieter Lechner; Quality Software
Assembly Language For The Applesoft Programmer
by Roy Meyers and C.W. Finley; Addison Wesley
What's Where In The Apple
by William Lubert; Micro Ink

Keying In Applesoft Programs

BASIC programs are printed in this Book Of Softkeys in a format that is designed to minimize errors for readers who key in these programs. To understand this format, you must first understand the formatted **LIST** feature of Applesoft.

An illustration- If you strike these keys:

```
10 HOME:REMCLEAR  SCREEN
```

a program will be stored in the computer's memory. Strangely, this program will **not** have a LIST that is exactly as you typed it. Instead, the LIST will look like this:

```
     10  HOME : REM CLEAR SCREEN
```

Programs don't usually LIST the same as they were keyed in because Applesoft inserts spaces into a program listing before and after every command word or mathematical operator. These spaces usually don't pose a problem except in line numbers which contain REM or DATA command words. The space inserted after these command words can be misleading. For example, if you want a program to have a list like this:

```
     10  DATA  67,45,54,52
```

you would have to omit the space directly after the DATA command word. If you were to key in the space directly after the DATA command word, the LIST of the program would look like this:

```
     10  DATA    67,45,54,52
```

This LIST is different from the LIST you wanted. The number of spaces you key after DATA and REM command words is very important.

All of this brings us to the Hardcore COMPUTIST LISTing format. In a BASIC LISTing, there are two types of spaces: spaces that don't matter whether they are keyed or not, and spaces that **MUST** be keyed. The latter spaces are printed here as delta characters (▵). All other spaces in our BASIC LISTing are put there for easier reading and it won't matter whether you type them or not.

Keying In Hexdumps

Machine language programs are printed here as both source code and hexdumps. Only one of these formats need be keyed in to get a machine language program. Hexdumps are the shortest and easiest format to type in. To key in hexdumps, you must first enter the Monitor with **CALL -151** RETURN.

Now key in the hexdump exactly as it appears. If you hear a beep, you will know that you have typed something incorrectly and must retype that line.

When finished, return to BASIC by typing **E003G** RETURN. Remember to BSAVE the program with the correct filename, address and length parameters as given in the article.

Keying In Source Code

The source code portion of a machine language program is provided only to better explain the program's operation. If you wish to key it in, you will need the *S-C Assembler*. Without this assembler, you will have to convert the *S-C Assembler* directives (printed in Hardcore COMPUTIST # 17) to similar directives used by your assembler.

Apple LOGO

Logo Computer Systems, Apple Computer

How to Copy Apple LOGO
by Anne Rachel Gygi
(Hardcore COMPUTIST # 8, page 7)

Requirements:
Apple][Plus with 64K RAM
One disk drive with *DOS 3.3*
A sector editor such as *DiskEdit*
A Bit Copy program such as *Locksmith* or *Copy II Plus*
One blank disk

Apple LOGO is copy-protected by writing Track $1 in a non-standard format. A nibble-count technique is used on this track with a fixed number of $FFs being written between $D6. If a copy is made and the number of $FFs between $D6s on Track $1 is not the same as on the original disk, then the copy will not work.

Track $1 on the distributed disk has the following format:

Number of Bytes:	1	124	1	132	1
Value:	$D6	$FF	$D6	$FF	$D6
Rest of Track:	$FF				

The logic to read and analyze Track $1 is in the second boot load Track $Ø, Sector $ØA, which ultimately resides in memory at $4ØØØ—$4ØFF. There are two constants equal to the number of Sync Bytes ($FFs) between the three $D6s and these are loaded into memory locations $4ØCD (124 or $7C) and $4ØCE (132 or $84). These are bytes $7C and $7D on Track $Ø, Sector $ØA.

Unlocking the Turtle

The technique for copying *Apple Logo* involves making a sector edit to a bit copy so that the code to seek and read Track $1 is disabled. The necessary steps are outlined next.

1 Use a bit copier (*Locksmith 4.1*, *Nibbles Away*, etc.) to copy Tracks $Ø—$22 with no parameter changes. An error on Track $1 is OK.

2 Use a Sector-Editing program such as *DiskEdit* or *ZAP* from *Bag of Tricks* to make the following changes to the copied disk.

Track	Sector	Byte	To
Ø	$A	$13	$EA
Ø	$A	$14	$EA
Ø	$A	$15	$EA
Ø	$A	$22	$4C
Ø	$A	$23	$55
Ø	$A	$24	$4Ø
Ø	$A	$79	$EA
Ø	$A	$7A	$EA
Ø	$A	$7B	$EA

Don't forget to write the sector back to the disk.

The first set of changes at $13—$15 eliminates (NOPs) the branch to *Logo*'s RWTS at $3DØØ which seeks Track $1. The changes at $22—$23 cause a branch around the code which reads Track $1 and the final changes at $79—$7B eliminate (NOPs) the branch which would be taken if the nibble count is not correct.

The resulting disk can now be copied with any bit copier and *Apple Logo* will run because it now completely ignores whatever is on Track $1.

Wes Felty's APT for...

Ultima

Lots Of Ships

When you board a ship and leave land, your ship will often split into two ships. The second ship will attack you, but do not sink it. Instead, land your ship, exit it and board the second ship. Then sail it to another continent and exit the ship.

If you reboard the ship and set sail, it may split in two again. You can leave ships all over the world using this technique.

Arcade Machine

Broderbund Software, Inc

Softkey For The Arcade Machine
by Marco Hunter
(Hardcore COMPUTIST # 1Ø, page 9)

Requirements:
Apple with 48K
One disk drive, with *DOS 3.3*
One blank disk
Old Monitor ROM or modified F8 ROM
Super IOB
Arcade Machine

Although Broderbund Software is generally very thorough with their protection schemes, they left two holes in the protection on the *Arcade Machine*.

First of all, Tracks $Ø3—$11 are normal *DOS 3.3* tracks. These tracks store the various parts of the options available from the menu.

Secondly, although the main file is heavily protected on the disk (spiral protection and what not), once loaded into memory it is fairly clean.

Since the menu options take up Tracks $Ø3—$11 we have room for DOS (Tracks Ø—2), as well as room for the main file (Tracks $12—$22).

Because Track $11 is taken up by the menu options, we must change the location of our CATALOG. I chose to put it on Track $12.

Now we have a neat package (all on one disk) just like the original.

When we boot our deprotected version, DOS will load the main file which from then on will access the menu options on Tracks $Ø3—$11.

Here's how

1 First of all, boot up with a DOS that has the ability to INITialize a disk (a fast DOS is recommended).

2 Next, type in the Super IOB controller at the end of this article and SAVE it. (See the **Controller Saver** article.)

3 With the controller and Super IOB merged, execute the Super IOB program:

RUN

4 When Super IOB asks you if you want to format the target disk, **you must type a** [Y]. This formatting is necessary because the controller does a special format to the disk which puts the directory on Track $12 instead of $11.

5 Once Super IOB has copied Tracks $Ø3—$11, boot up the *Arcade Machine*:

PR#6

6 Get into the Monitor when the main menu appears.

7 Move the main file and sensitive memory to safety:

```
2000<9600.B600M
8000<B600.C000M
8A00<0.900M
```

8 Boot the disk that Super IOB formatted (you should get a *?FILE NOT FOUND* error, but don't worry):

6[CTRL]P

9 Enter the Monitor:

CALL -151

10 Put a patch just before the beginning of the *Arcade Machine* code:

```
8FD:4C 00 93
```

11 Type in this routine that moves the memory back to where it used to be:

```
9300:  A2  00  BD  00  20  9D  00  96
9308:  E8  D0  F7  EE  04  93  EE  07
9310:  93  AD  04  93  C9  40  D0  EA
9318:  BD  00  80  9D  00  B6  E8  D0
9320:  F7  EE  1A  93  EE  1D  93  AD
9328:  1D  93  C9  C0  D0  EA  BD  00
9330:  8A  9D  00  00  E8  D0  F7  EE
9338:  30  93  EE  33  93  AD  33  93
9340:  C9  09  D0  EA  4C  00  08
```

You can check your typing by comparing it to the following disassembly:

```
9300    A2 ØØ     LDX #$ØØ
93Ø2-   BD ØØ 2Ø  LDA $2ØØØ,X
93Ø5-   9D ØØ 96  STA $96ØØ,X
93Ø8-   E8        INX
93Ø9-   DØ F7     BNE $93Ø2
93ØB-   EE Ø4 93  INC $93Ø4
93ØE-   EE Ø7 93  INC $93Ø7
9311-   AD Ø4 93  LDA $93Ø4
9314-   C9 4Ø     CMP #$4Ø
9316-   DØ EA     BNE $93Ø2
9318-   BD ØØ 8Ø  LDA $8ØØØ,X
931B-   9D ØØ B6  STA $B6ØØ,X
931E-   E8        INX
931F-   DØ F7     BNE $9318
9321-   EE 1A 93  INC $931A
9324-   EE 1D 93  INC $931D
9327-   AD 1D 93  LDA $931D
932A-   C9 CØ     CMP #$CØ
932C-   DØ EA     BNE $9318
932E-   BD ØØ 8A  LDA $8AØØ,X
9331-   9D ØØ ØØ  STA $ØØØØ,X
9334-   E8        INX
9335-   DØ F7     BNE $932E
9337-   EE 3Ø 93  INC $933Ø
933A-   EE 33 93  INC $9333
933D-   AD 33 93  LDA $9333
934Ø-   C9 Ø9     CMP #$Ø9
9342-   DØ EA     BNE $932E
9344-   4C ØØ Ø8  JMP $Ø8ØØ
```

12 Save the main file:

BSAVE ARCADE ▵MACHINE, A$8FD, L$8DØ4

13 Return to BASIC:

FP

14 Type in this short greeting program:

1Ø PRINT CHR$ (4) "BRUN ARCADE ▵MACHINE"

15 Save it:

SAVE HELLO

The *Arcade Machine* is now *COPYA*able.

The backside (with all the sample games) can be copied with any good bit copier.

If you don't have a bit copier, you can insert a **POKE 47426,24** into Line 1Ø of *COPYA* to copy it.

One final note: The options from the *Arcade Machine* were all written in Applesoft. If you RESET into the Monitor and type ctrl C to get into BASIC after choosing one of the options, you can then list the file.

controller

```
410 GOSUB 80 : HOME :A$ = "FORMATING" : FLASH : GOSUB 450 : NORMAL : POKE
     44033 ,18 : POKE 44703 ,18 : POKE 44764 ,18 : POKE 42347 ,96
415 POKE 43364 ,255 : PRINT : PRINT CHR$ (4 ) "INIT^HELLO,V"VL",S"S2
     ",D" D2 :VL = Ø : RETURN
1ØØØ REM ARCADE MACHINE
1Ø1Ø TK = 3 :ST = Ø :LT = 18 :CD = WR : POKE 47426 ,24 : GOSUB 112Ø
1Ø2Ø T1 = TK : GOSUB 49Ø
1Ø3Ø GOSUB 43Ø : GOSUB 1ØØ :ST = ST + 1 : IF ST < DOS THEN 1Ø3Ø
1Ø4Ø IF BF THEN 1Ø6Ø
1Ø5Ø ST = Ø :TK = TK + 1 : IF TK < LT THEN 1Ø3Ø
1Ø6Ø GOSUB 49Ø :TK = T1 :ST = Ø
1Ø7Ø GOSUB 43Ø : GOSUB 1ØØ :ST = ST + 1 : IF ST < DOS THEN 1Ø7Ø
1Ø8Ø ST = Ø :TK = TK + 1 : IF BF = Ø AND TK < LT THEN 1Ø7Ø
1Ø9Ø IF TK < LT THEN 1Ø2Ø
11ØØ HOME : PRINT : PRINT "DONE^WITH^COPY" : END
112Ø POKE 44Ø33 ,17 : POKE 447Ø3 ,17 : POKE 44764 ,17 : POKE 42347 ,76
113Ø CD = RD :SO = S2 :DV = D2 : GOSUB 8Ø :TK = 18 :ST = Ø : GOSUB 43Ø :
     GOSUB 1ØØ
114Ø POKE 9985 ,18 : FOR A = 1ØØ52 TO 1Ø115 : POKE A ,Ø : NEXT :CD = WR
     : GOSUB 8Ø : GOSUB 1ØØ :TK = 3 : RETURN
```

The Artist

Sierra On-Line

Softkey for The Artist
by Walt Campbell
(Hardcore COMPUTIST # 8, page 12)

Requirements:

Apple][Plus or Apple //e, 48K
At least one disk drive
The Artist master disk
A blank disk
FID or *COPYA*

The Artist is the graphics package used to develop some of Sierra On-Line's biggest selling games, such as *Crossfire* and *Mouskattack*. That is what the introduction to the program claims and after using the package for quite awhile, I can believe it. It contains two separate but compatible screen drawing routines, a zoom lens type bit editor, a 21-color fill routine, shape table construction and editor utilities, and some nifty high-speed byte-shape design and move routines. All in all, an exceptional set of graphics tools.

The problem, as with all of Sierra On-Line's programs, is that it's copy-protected. The copy-protection scheme is similar to the one outlined by Dan Price in the **Screenwriter II softkey** (Book Of Softkeys volume I). The program uses direct disk access to check the format of the disk tracks for embedded keys. If the keys are not present, it clears memory and reboots the disk. The program is otherwise pretty much box-stock *DOS 3.3* and can be easily copied using *FID* or *COPYA* from the *DOS System Master* disk.

I jumped into the project of deprotecting the disk after reading Mr. Price's softkey, feeling confident that in a short time I would have identified the subroutine that checked the disk, bypassed it, and have a 'clean' unprotected copy. Eight hours later I was no closer to a copy and much more frustrated. I took a break from the computer and skimmed through my back issues of Hardcore COMPUTIST looking for a hint or clue that might aid me in 'cracking' this code.

In Ray Darrah's article **Boot-code Tracing Pest Patrol** (see Book Of Softkeys Volume I), I found a discussion of concealed code and disguised jumps in machine language programs. This 'turned on a light' and I returned to the project with new optimism. Two hours later I had my unprotected copy! Once discovered, the procedure is (naturally!) simple and can be done in fifteen minutes. Before outlining the procedure I would like to explain the protection scheme more fully in hopes of aiding other enterprising hackers and maybe saving them hours of time. Those of you not interested in the procedures used to break the copy protection can skip to the fix below.

The copy protection method used on *The Artist* is similar to the one used on *Screenwriter II*, but the author of the program not only uses concealed jumps and stack manipulation to hide return addresses, he has a particularly sneaky portion of code that actually rewrites itself before it checks the disk, and then rewrites itself after it's done to conceal its existence.

After much experimentation, I isolated the disk checking routine to a range of code in the main menu program. This code plays some tricks with branch instructions and forces branches to what seems to be the middle of other valid instructions. It pushes addresses on the stack and executes a machine language return to jump to those addresses, etc. This kind of code has become pretty standard in copy and code protection but is frustrating, nonetheless.

After digging through this mess I came to a section of code that seemed to make no sense whatsoever and ended by jumping to an area of memory that I knew contained no program code! I started inserting breaks in the code to try and isolate the offending section and, BINGO! I found a short section of disguised code that actually rewrote successive bytes of the program! This type of protection was new to me and may serve to illustrate further the possibilities available to machine code programmers in code manipulation.

The program loads the Y-Register with a value of $29, indexes to an address $29 bytes away from itself, and performs an exclusive 'OR' operation (EOR) with a value of $8A on the code at this new location.

This transforms a seemingly meaningless section of code to a detailed byte-check of the tracks on the disk and checks for the embedded keys. If the proper format keys are not found, it clears memory and reboots the disk. If it finds the proper keys, it rewrites itself and returns to the main section of code! Once I got this far in the analysis the fix became obvious and extremely simple.

THE FIX:

In order to remove this protection it is necessary to first copy the disk with either the *COPYA* or the *FID* programs. *The Artist* master disk uses a binary *Hello* program, so if you use *FID* to copy the programs, you must either INIT your blank disk to BRUN the *Hello* program (as outlined in the Dan Price's **Screenwriter Softkey**) or use a *DOS* utility, such as *ProntoDOS*, to change the disk boot program. Using a modified *DOS*-like *ProntoDOS* also will noticeably speed up program disk access and the loading and saving of hi-res pictures.

Once you have prepared your new program disk and it contains all the files from the original master disk, follow the series of steps listed below:

1 Boot up a normal *DOS 3.3* disk:

```
PR#6
```

2 Insert the copy of *The Artist* in your disk drive.

3 Unlock the *MAIN MENU* program so it can be modified:

```
UNLOCK MAIN MENU
```

4 Load the *MAIN MENU* program:

```
BLOAD MAIN MENU
```

6 Enter the Monitor:

```
CALL -151
```

5 Modify Address $4257 from $8A to $57 and Address $4662 from $B9 to $60:

```
4257:57
4662:60
```

6 Save the modified *MAIN MENU* program:

```
BSAVE MAIN MENU,A$4000,L$4D
```

7 Finally, relock the now modified program:

```
LOCK MAIN MENU
```

Alternatively, if you used *COPYA* to copy the disk, the changes can be made directly to the disk with a sector editor. On my copy of *The Artist*, the first change to make was at Track $05, Sector $0C, Byte $5B. I changed this byte from $8A to $57. The second change is at Track $05, Sector $08, Byte $66. This byte needs to be changed from $B9 to $60. This probably won't work on a *FID* copy because the sector allocation may be different.

THAT'S IT!! You now have a copy of *The Artist* that is unprotected and *COPYA*able!

Bank Street Writer

Bank Street Writer

Softkey For Bank Street Writer
by Earl Taylor & Steve Morgan
(Hardcore COMPUTIST # 1Ø, page 12)

Requirements:
Apple][, Apple][Plus, //e (minimum 48K and Applesoft)
One disk drive and *DOS 3.3*
Blank disks

The *Bank Street Writer* is a word-processing program particularly suited to the beginner. As such it includes many features to prevent confusion and accidents which might be experienced by the computer novice. In developing this softkey we have attempted to retain as many of these features as possible.

The original disk is able to operate with several different configurations of hardware. It requires (and checks for) a minimum of 48K of motherboard RAM. In addition, the presence of Applesoft is verified. If a 16K RAMcard is installed in Slot Ø, Applesoft will be placed there if necessary. If Applesoft is in Motherboard ROM, the extra 16K will be used to increase the user's text memory allowing about 3,2ØØ words instead of 1,3ØØ without the card. If installed into an Apple //e, the program will use the //e's extra memory and the (open-apple), (closed-apple) keys along with the four cursor movement keys ←→↑↓.

Extensive error-trapping is performed and hitting RESET is totally non-destructive to the user's text, requiring only a press of the RETURN key to carry on.

After experiencing some difficulty in getting the softkey version to properly operate the disk drive, we came upon a feature not described in the manual. The program actually prevents the user from accidentally or otherwise INITing or SAVEing files to the master disk (something that our instincts prevented us from trying!).

We have made an effort to maintain as many of these user-proof features as possible in our softkey. The unlocked softkey version supports all but the following features of the original:

1) At least 48K of RAM and Applesoft must be resident in the machine at the time the softkey version is booted.

> Our start-up program is in Applesoft. It is possible to enter the program as an Integer BASIC file and include a few statements to load Applesoft into the RAM card. The program will then work normally. All the users we know have Applesoft in ROM on the Motherboard; therefore, we felt this was not a serious limitation.

2) The original boots VERY QUICKLY compared to the softkey version.

> *Bank Street Writer* (BSW) uses a powerful bulk-loading scheme to bring its files into memory. This scheme depends on the format of the data on the disk and, of course, that is what we want to change. If you're not the kind who likes to wait around, we strongly recommend the use of one of the fast DOSs such as *Diversi-DOS*, *Hyper-DOS*, *ProntoDos*, etc.

The Boot Process

The following paragraphs describe the boot activity of the original disk as an insight into the softkey method. You may, of course, simply follow **'The Softkey Steps'** if you wish.

BSW, like all disks, has a Track Ø, Sector Ø that must be readable by the disk controller card's firmware at $C6ØØ (assuming the card is in Slot 6). This data is loaded into $8ØØ—$8FF and executed. At this point, more data is read from Track Ø and placed into memory at $1ØØØ—$17ØØ. Execution is passed to the code starting at $14ØØ.

At this point the program verifies that at least 48K of RAM is present, printing a message and stopping if it finds less. If all is well, the BSW DOS is loaded from Tracks $1 and $2 into $96ØØ—$BFFF.

Next, the type of hardware configuration is determined and an identifying character is printed in the upper left corner of the screen. There are four possible configurations which may occur. A value of **Ø, 1, 2, or 3** is stored in memory location $1F depending on the type of hardware.

Value	=	**Screen character**	: representation
Ø	=	**Ø**	: No Applesoft and no ramcard in Slot Ø.
1	=	**4**	: Applesoft in ROM or ramcard and 48K.
2	=	**6**	: Applesoft in ROM and ramcard in Slot Ø
3	=	**E**	: Apple //e.

If no motherboard ROM Applesoft is found but a ramcard is detected, the type-checking code automatically loads Applesoft from Tracks 3, 4 and 5 into the card and activates it. For all intents, this is the same configuration as a 48K Apple][Plus (*$1F = 1*).

If **$1F = Ø** after the type-check, the program informs the user that Applesoft is required and stops. For the other configurations more data is loaded into memory as follows:

$1F=	Tracks loaded...	into memory
1	$ØE, $ØF, $1Ø	$6ØØØ to $8FFF
2	$Ø6, $Ø7, $Ø8 $Ø9	$DØØØ to $FFFF $8BØØ to $9AFF
3	$ØA, $ØB, $ØC $ØD	$DØØØ to $FFFF $8BØØ to $9AFF

Next, hi-res graphics page two is displayed and filled with the title page from Tracks $18—$19. More data is read into $4ØØ—$7FF from Track $1A. The drive head is then positioned to Track $11 and the *BSW DOS* is cold-started with a jump to $9D84.

Track $11 contains a file directory like a normal disk. The *BSW DOS* loads in an Applesoft greeting program at $8ØØ and RUNs it. The program filename is **A** followed by 7 backspace characters and 22 spaces. When we later modify normal DOS to read the catalog, this filename will not appear since the backspaces cause the visible characters to be overwritten by the spaces when printed to the screen.

Program **A** first detects if the ESC key has been pressed during the boot. If so, the *UTILITY* program is run (more on this later). Otherwise, **A** loads two other small binary files. One is called *Ø$3Ø1* which loads at $3Ø1 and the other is *INIT*, loaded at $2DØ. Once these are loaded, **A** continues the boot with a CALL 2Ø48. This begins execution of the code loaded earlier at $4ØØ.

Another hardware-dependent load is performed next as follows:

$1F=	Tracks loaded	into memory
1	$1F $2Ø, $21 $22	$9ØØØ to $9BFF $25ØØ to $3CFF $34ØØ to $3FFF
2	$1B, $1C	$Ø8ØØ to $1FFF
3	$1D, $1E	$Ø8ØØ to $1FFF

At this point, the disk access is finally completed and the drive is shut off. The next statement to be executed is at $525. This code does some initialization and patching and then starts the program with an indirect jump through $0000 to either $6009 for hardware type **1** ($1F = 1) or $098F for types **2** and **3**.

The Protection

The copy-protcction on the BSW disk consists essentially of changes to address and data prologue and epilogue bytes on some tracks and non-standard encoding and sectoring (for fast load) on other tracks.

Half-tracks, track-arcing, track synchronizing, and nibble counting do not seem to have been used. The disk can be bit-copied but seems to be speed sensitive which might indicate that the original was written at slower than normal speed.

The Softkey

To obtain an unprotected version of BSW, the original is allowed to load the required ranges of memory under the control of a small machine language program. At specific points in the boot, the program breaks into the monitor.

A slave DOS is then booted and used to save the files in the normal *DOS 3.3* format.

Finally, a BASIC program is written to emulate the functions of the original boot process, ending with a **CALL 1317** ($525) to do the initializing and start the program.

In an effort to make things a little more sensible, notes are included with the softkey steps. There are quite a number of files to capture and many machine language patches to perform, so we suggest you ***take your time and double check any typed-in code.***

1 Boot *DOS 3.3* into your system.

2 Enter the Monitor and change the VTOC buffer (explained later):

```
CALL -151
B3BF:A5
```

3 Return to BASIC and clear the program memory:

```
CTRL C
FP
```

4 Initialize a disk with this modified VTOC:

```
INIT HELLO
```

5 Move the disk][controller ROM into RAM:

```
3600<C600.C6F7M
```

6 Enter the following machine language

```
36F8:  A9  Ø5  8D  21  Ø8  A9  37  8D
37ØØ:  22  Ø8  4C  Ø1  Ø8  AØ  ØØ  B9
37Ø8:  2D  37  99  F8  13  C8  CØ  Ø5
371Ø:  DØ  F5  8D  84  14  8D  86  14
3718:  8D  AØ  17  A9  4B  8D  B6  14
372Ø:  A9  32  8D  25  15  A9  37  8D
3728:  26  15  4C  ØØ  14  A9  Ø1  85
373Ø:  1F  6Ø  A9  4C  8D  25  Ø5  A9
3738:  59  8D  26  Ø5  A9  FF  8D  27
374Ø:  Ø5  4C  84  9D  A9  2C  8D  CA
3748:  14  8D  D1  14  4C  ØØ  14
```

Check your entry against the following disassembly:

```
36F8-   A9 Ø5       LDA #$Ø5
36FA-   8D 21 Ø8    STA $Ø821
36FD-   A9 37       LDA #$37
36FF-   8D 22 Ø8    STA $Ø822
37Ø2-   4C Ø1 Ø8    JMP $Ø8Ø1
37Ø5-   AØ ØØ       LDY #$ØØ
37Ø7-   B9 2D 37    LDA $372D,Y
37ØA-   99 F8 13    STA $13F8,Y
37ØD-   C8          INY
37ØE-   CØ Ø5       CPY #$Ø5
371Ø-   DØ F5       BNE $37Ø7
3712-   8D 84 14    STA $1484
3715-   8D 86 14    STA $1486
3718-   8D AØ 17    STA $17AØ
371B-   A9 4B       LDA #$4B
371D-   8D B6 14    STA $14B6
372Ø-   A9 32       LDA #$32
3722-   8D 25 15    STA $1525
3725-   A9 37       LDA #$37
3727-   8D 26 15    STA $1526
372A-   4C ØØ 14    JMP $14ØØ
372D-   A9 Ø1       LDA #$Ø1
372F-   85 1F       STA $1F
3731-   6Ø          RTS
3732-   A9 4C       LDA #$4C
3734-   8D 25 Ø5    STA $Ø525
3737-   A9 59       LDA #$59
3739-   8D 26 Ø5    STA $Ø526
373C-   A9 FF       LDA #$FF
373E-   8D 27 Ø5    STA $Ø527
3741-   4C 84 9D    JMP $9D84
3744-   A9 2C       LDA #$2C
3746-   8D CA 14    STA $14CA
3749-   8D D1 14    STA $14D1
374C-   4C ØØ 14    JMP $14ØØ
```

7 Save the boot ROM and program:

BSAVE CODEBREAK,A$36ØØ,L$14F

How CODEBREAK Works

The *CODEBREAK* program was developed by boot code tracing the original disk. It operates as follows:

36ØØ . 36F7 Copied from the disk controller card, reads in the track.

3732 . 3743 Places JMP ($4C) to $FF59 at $Ø525 which allows us to break into the Monitor after the last files are loaded. The boot is continued by cold-starting the *BSW DOS* with a JMP $9D84.

3744 . 374E Used later to store $2C (a harmless BIT instruction) on top of two JSR instructions used to load the hi-res page two graphic. We will change the JMP $14ØØ at $372A to a JMP $3744 after we have saved the logo. We can then use the space from $4BØØ—$5AFF for the third portion of the type **2** and **3** loads.

Whew! Let's Get Some Files

8 Put the BSW disk in the drive and execute *CODEBREAK*:

36ØØG

9 When the drive stops, make the following machine language patch:

```
1300: 2C 81 C0 20 55 13 F0 06
1308: 2C 80 C0 4C 15 13 20 3E
1310: 13 F0 1C EA EA
```

A **13ØØL** should produce:

```
1300-   2C 81 C0    BIT $C081
1303-   20 55 13    JSR $1355
1306-   F0 06       BEQ $130E
1308-   2C 80 C0    BIT $C080
130B-   4C 15 13    JMP $1315
130E-   20 3E 13    JSR $133E
1311-   F0 1C       BEQ $132F
1313-   EA          NOP
1314-   EA          NOP
1315-   A2 01       LDX #$01
```

etc...

Because our softkey requires Applesoft to be installed before running, some minor changes to the BSW type-checking routine at $1300 are needed to maintain maximum compatibility. The BSW type-checker normally first checks for a RAMcard by writing and verifying that every bit pattern can be stored at $D000. This would clobber Applesoft which might have been loaded there by the user before running the softkey version. To handle this situation, the above code first enables motherboard ROM with a **BIT $C081** and calls the BSW 'check for Applesoft' routine at $1355. This routine returns with the zero flag set if Applesoft is found. If it is found, we go to $130E where we check for a RAMcard via the **JSR $133E**. If a RAMcard is found, the code branches to the //e-checker at $132F. That code will exit with a 'type 2' if a //e is not found or with a 'type 3' if it is found.

If Applesoft is not found in motherboard ROM, execution falls through to $1308. We couldn't have gotten here with the softkey version if Applesoft were not in Slot 0, so it is re-activated and the code exits indicating a 'type 1'. If the ramcard check at $130E fails, the routine also exits as a 'type 1'.

Let's Continue

10 Move the BSW code out of the way so our slave disk can boot:

```
1400<9000.9BFFM
```

11 Put the slave disk in the drive and boot it:

```
C600G
```

12 We will now save as many files as possible:

```
BSAVE TYPECHECK,A$1300,L$69
BSAVE BSW.LOGO,A$4000,L$2000
BSAVE BSW.48K.1,A$6000,L$3000
BSAVE BSW.48K.2,A$2500,L$1B00
BSAVE BSW.48K.3,A$1400,L$AA6
```

13 We are ready to move on to the 'type 2' files. So hop into the Monitor and load *CODEBREAK*:

```
CALL -151
BLOAD CODEBREAK
```

14 Prepare *CODEBREAK* for 'type 2' files:

```
372E:02
```

15 Enable the 'hi-res graphic load disabler':

```
372B:44 37
```

16 Put the BSW disk in the drive and execute the modified *CODEBREAK* program:

```
3600G
```

Remember, the hi-res graphics load has been disabled so don't be alarmed when you don't see the logo appear.

17 Move the file out of the way so we can boot:

```
2800<800.1FFFM
```

18 Boot the slave disk again:

```
C600G
```

19 Save all of the 'type 2' files:

```
BSAVE BSW.64K.1,A$6000,L$3000
BSAVE BSW.64K.2,A$2800,L$1800
BSAVE BSW.64K.3,A$4B00,L$FA6
```

20 To get the //e files, enter the Monitor, load *CODEBREAK*, and tell it we want 'type 3' files:

```
BLOAD CODEBREAK
CALL -151
372E:03
```

21 Prevent the hi-res graphic load:

```
372B:44 37
```

22 Put the BSW original in the drive and start up *CODEBREAK*:

```
3600G
```

23 Move this file out of the slave disk's boot path:

```
2800<800.1FFFM
```

24 Put the slave disk in the drive and boot it by typing:

```
C600G
```

25 Now save the //e files:

```
BSAVE BSW.//E.1,A$6000,L$3000
BSAVE BSW.//E.2,A$2800,L$1800
BSAVE BSW.//E.3,A$4B00,L$FA6
```

What's Next?

We must now capture the initialization code which resides in pages $5—$7. This is not as straightforward as the other files because this memory range is part of the page 1 text screen. When we jump to the Monitor at $FF59 it immediately begins to print its prompt and scroll the screen. This action quickly destroys the BSW data stored there.

The data must be moved out of the text page before anything is printed. We could modify *CODEBREAK* to move the data before jumping to the Monitor, but we hate typing in code especially when there is an interesting alternative. BSW short-circuits the character switch pointer ($36—$37) by pointing it to an RTS instruction at $FF58 just before it cold-starts its DOS. This is necessary to prevent its DOS from destroying the data in the text page, since DOS normally prints a few prompts and carriage returns to the text screen as it starts up.

We will take advantage of this by entering the Monitor at $FF62 instead of $FF59. This will prevent the Monitor from fixing the character switch and give us time to move the data manually. Since no characters can be printed to the screen, we will be 'blind' for a moment so type carefully.

26 Tell *CODEBREAK* to turn off the drive and JuMP into the Monitor without resetting the character output switch:

```
BLOAD CODEBREAK
CALL -151
3732:2C E8 C0 4C 62 FF
```

27 Put the BSW original in the drive and boot it with:

```
3600G
```

You will not see any of the characters you type, so be careful. If you think you made a mistake, type CTRL X and start over.

28 Move the file out of the way (be careful):

```
2500<500.7FFM
```

29 Reset the character output switch and enter the Monitor:

```
FF59G
```

You should now see the Monitor prompt. If you type **2500L**, you should see:

```
2500-   20 86 04    JSR $0486
2503-   A5 1F       LDA $1F
2505-   0A          ASL
```

etc...

If this is what you see, you've got it! If not, try Steps 27 through 29 again.

30 Put the slave disk back in the drive and boot it:

```
C600G
```

31 Save the code beginning at $0525:

```
BSAVE BSW.$525.$7FF,A$2525,L$2DB
```

Getting The CATALOGable Files

The next step is to capture the CATALOGable files from the BSW original. In order to do this, the RWTS (Read or Write a Track/Sector) routine of our slave DOS must be modified to handle the non-standard prologue and epilogue bytes of the BSW disk.

32 Enter the Monitor and make the necessary changes to DOS:

```
CALL -151
B8E7:D4
B8F1:D5
B8FC:D6
B935:D7
B938:18 60
B955:A5
B95F:96
B96A:BF
B991:9A
B994:18 60
```

A CATALOG of the BSW original should reveal the names of the files mentioned earlier. Notice the extra blank line at the top of the listing. This is where the **A** filename was printed. Since it is a tedious procedure to change DOS like this, it is convenient that we can place all these files into memory at the same time.

33 Load the files:

```
BLOAD INIT,A$6300
BLOAD O$301,A$7301
BLOAD PASSWORD!,A$8000
LOAD UTILITY
```

UTILITY contains two lines that call a machine language subroutine. This routine is used by *UTILITY* to change the BSW DOS from non-standard to standard and back again depending on whether the BSW disk or the data disk is being accessed. *UTILITY* calls the routine through the now famous ampersand (**&**) vector.

Happily, our slave DOS will be the same format for both the BSW softkey disk and our data disks. If we remove the **&** calls, *UTILITY* will work fine with our new unlocked BSW disk.

34 Prepare to edit line 12:

```
HOME: POKE 33,33: LIST 12
```

35 Using the ESC keys, and right arrow → key, remove from this line the **&1** and **&0**.

36 Remove the **&1** and **&0** from line 98:

```
LIST 98
```

37 To save the files, we must switch back to standard DOS sector marker bytes. Insert the slave disk and type:

```
CALL -151
B8E7:D5
B8F1:AA
B8FC:AD
B935:DE
B955:D5
B95F:AA
B96A:96
B991:DE
```

38 We can now save the files in standard format:

```
SAVE UTILITY
BSAVE INIT,A$6300,L$18
BSAVE O$301,A$7301,L$CF
BSAVE PASSWORD!,A$8000,L$180
```

We have finally obtained all the required files from the BSW original disk. The remaining tasks are:

1) Write a program to emulate the boot process.
2) Write two small machine language DOS patches.

The Emulation Program

Many of the BSW files were relocated before they were saved to our slave disk. These files can be loaded back into position simply by specifying the Address parameter of the BLOAD command.

However, there are four files that must be given special consideration.

The *BSW.48K.3* file is loaded by the BSW original between $9000 and $9BFF. This would destroy our slave DOS's file buffer even if we set MAXFILES 1. A close examination of the file reveals that it contains no meaningful data past $9AA5 which is the bottom of the first DOS file buffer. When we saved the file in Step 12, the length parameter was shortened to allow us to BLOAD the file without overwriting the file buffer.

For types 2 and 3, the same problem occurs with *BSW.64K.3* and *BSW.//e.3* and the same solution has been used.

The *BSW.64K.2* and *BSW.//e.2* files both need to be loaded in the range $800—$1FFF. This is normally Applesoft program space and would conflict with our *HELLO* program.

The solution is to cause DOS to load our *HELLO* program into the unused space starting at $2000 instead of the normal $800. Since our program will be fairly short, we can load it at $2000 for all three hardware types and not conflict with *BSW.48K.2* at $25ØØ.

39 After booting the slave disk, type in the following Applesoft program. When you have checked for and corrected any errors, SAVE it with **SAVE HELLO** RETURN.

```
1ØØ IF PEEK (1Ø4 ) <> 32 THEN POKE 8192 ,Ø : POKE 1Ø4 ,32 : PRINT CHR$
      (4 ) "RUN HELLO"
11Ø TEXT : HOME : NORMAL : IF PEEK (49152 ) = 155 THEN POKE 49168 ,Ø :
      POKE 1Ø4 ,8 : PRINT CHR$ (4 ) "RUN UTILITY"
12Ø IF PEEK (49152 ) = 197 OR PEEK (49152 ) = 229 THEN POKE 49168 ,Ø :
      PRINT CHR$ (4 ) "FP"
13Ø POKE 49168 ,Ø : PRINT CHR$ (4 ) "MAXFILES1"
14Ø PRINT CHR$ (4 ) "BRUN TYPECHECK,A$13ØØ"
175 PRINT CHR$ (4 ) "BLOAD BSW.LOGO,A$4ØØØ"
18Ø POKE 49234 ,Ø : POKE 49237 ,Ø : POKE 49232 ,Ø : POKE 49239 ,Ø
19Ø PRINT CHR$ (4 ) "BLOAD O$3Ø1,A$3Ø1"
2ØØ PRINT CHR$ (4 ) "BLOAD INIT,A$2DØ"
21Ø PRINT CHR$ (4 ) "BLOAD WRITEPROTECT,A$B7ØØ"
22Ø IF PEEK (31 ) > 1 THEN 26Ø
23Ø PRINT CHR$ (4 ) "BLOAD BSW.48K.1,A$6ØØØ"
24Ø PRINT CHR$ (4 ) "BLOAD BSW.48K.2,A$25ØØ"
25Ø PRINT CHR$ (4 ) "BLOAD BSW.48K.3,A$9ØØØ" : GOTO 34Ø
26Ø PRINT PEEK ( - 16255 ) + PEEK ( - 16255 )
27Ø IF PEEK (31 ) = 3 THEN 31Ø
28Ø PRINT CHR$ (4 ) "BLOAD BSW.64K.1,A$DØØØ"
29Ø PRINT CHR$ (4 ) "BLOAD BSW.64K.2,A$8ØØ"
3ØØ PRINT CHR$ (4 ) "BLOAD BSW.64K.3,A$8BØØ" : GOTO 34Ø
31Ø PRINT CHR$ (4 ) "BLOAD BSW.//e.1,A$DØØØ"
32Ø PRINT CHR$ (4 ) "BLOAD BSW.//e.2,A$8ØØ"
33Ø PRINT CHR$ (4 ) "BLOAD BSW.//e.3,A$8BØØ"
34Ø POKE 448Ø2 ,234 : POKE 448Ø3 , 234 : POKE 448Ø4 ,234 : POKE 44723
      ,4 : POKE 216 ,255
345 PRINT CHR$ (4 ) "BLOAD PATCH,A$BFC8"
35Ø PRINT CHR$ (4 ) "BRUN BSW.$525.$7FF,A$525"
```

This program does not create any variables. In particular, the CHR$(4) must be used repeatedly rather than defining a **D$ = CHR$(4)** since strings would be stored below HIMEM and conflict with files to be loaded there. The program length has been minimized by deleting unnecessary spaces in DOS commands since it must fit between $2ØØØ and $24FF.

The *HELLO* program begins by determining where it has been loaded. Location 1Ø4 ($68) contains the high byte of the start of Applesoft program pointer which is normally $Ø8. If this value is not 32 ($2Ø) then the program pokes a zero at 8192 ($2ØØØ) and changes the pointer to point there. A DOS command to RUN HELLO then reloads the program at $2ØØØ and starts it running. Of course, the second time through, 1Ø4 will contain 32 and the program carries on.

Line **110** tests for the ESC key waiting at the keyboard and, if found, resets the start of Applesoft pointer to $0800 and RUNs *UTILITY*. As a handy exit, Line **120** checks for upper or lowercase **e** at the keyboard and, if found, gives the DOS FP command to reset the pointer to $0800, clear the program and stop. This is convenient when you want to examine files on the disk; just hit E during the boot, then load whatever you wish to look at.

Line **130** clears the keyboard of any other keys and then sets MAXFILES 1. Line **140** runs *TYPECHECK* to determine the hardware configuration and lines **150—190** load the files common to all three hardware types. Line **160** sets the soft-switches to display the hi-res page 2 logo. The *WRITEPROTECT* file in Line **190** is created below.

Line **200** checks location 31 ($1F) and skips to line **140** if ' 'type 2' or 'type 3' has been determined by *TYPECHECK*. For 'type 1', lines **210—230** load the ...*48K* files and GOTO 320 to finish execution.

Line **240** write-enables the RAMcard so it may be loaded for types 2 and 3. Line **250** directs flow to either the ...*64K* or ...*//e* file loading.

Line **320** pokes three NOPs into DOS at 44802 ($AF02). This skips putting DOS onto data disks initialized by BSW and, although not done by the original BSW, the 4 poked at 44723 frees that extra space for data storage. The Applesoft ONERR flag 216 ($D8) is set to prevent DOS from issuing its own error messages since the program handles these itself.

Another small piece of code is loaded into DOS in line **330** and the initialization program is run to start the BSW program.

The Write-Protect Patch

There is a small section of code at $B700 in the *BSW DOS* which is called whenever a write operation is about to be performed. This code reads the catalog track of the disk in the selected drive and checks for the non-standard disk bytes of the original BSW disk. If it finds this pattern, the carry flag is cleared, location $9AAC is loaded with a $4D and the routine returns to the caller. If the pattern is not found, the carry flag is set and the routine returns.

We originally thought this might be some kind of copy protection. It turned out to be a routine to prevent the user from writing to the original disk when he should not. Simply putting a write protect tab on the disk would provide similar protection, but this disk is likely to be used by children and it is easier to remove the write-protect tab than to change the format of the disk (as we have seen). In addition, the *UTILITY* program is used to customize BSW to the user's needs. It stores the initialization data on the BSW original disk and the write-protect tab would need to be removed for this operation.

It seemed logical to find a way to support this valuable feature. DOS maintains a buffer at $B3BB which contains the VTOC (Volume Table Of Contents) for the most recently read disk. This buffer is read from Track $11, Sector $Ø on any standard DOS disk. It is also written to a disk when that disk is INITed.

Not all bytes in the VTOC are used by DOS. Before we INITed our slave disk, we changed one of these unused bytes ($B3BF) to an $A5. This byte will be used by our own routine at $B7ØØ to determine if the disk about to be written is our softkey version of BSW.

40 Enter the Monitor and type the code that implements this feature on the softkey disk:

```
CALL -151
B7ØØ:  AØ Ø9 B9 EB B7 99 3D B7
B7Ø8:  B9 47 B7 99 EB B7 88 1Ø
B71Ø:  F1 AC C1 AA AD C2 AA 2Ø
B718:  B5 B7 BØ 1Ø AC BF B3 CØ
B72Ø:  A5 FØ Ø3 38 BØ Ø6 A9 4D
B728:  8D AC 9A 18 A9 ØØ 8D BF
B73Ø:  B3 AØ Ø9 B9 3D B7 99 EB
B738:  B7 88 1Ø F7 6Ø ØØ ØØ ØØ
B74Ø:  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
B748:  11 ØØ FB B7 BB B3 ØØ ØØ
B75Ø:  Ø1
```

41 Disassemble and check this code

```
B7ØØL

B7ØØ-   AØ Ø9      LDY #$Ø9
B7Ø2-   B9 EB B7   LDA $B7EB,Y
B7Ø5-   99 3D B7   STA $B73D,Y
B7Ø8-   B9 47 B7   LDA $B747,Y
B7ØB-   99 EB B7   STA $B7EB,Y
B7ØE-   88         DEY
B7ØF-   1Ø F1      BPL $B7Ø2
B711-   AC C1 AA   LDY $AAC1
B714-   AD C2 AA   LDA $AAC2
B717-   2Ø B5 B7   JSR $B7B5
B71A-   BØ 1Ø      BCS $B72C
B71C-   AC BF B3   LDY $B3BF
B71F-   CØ A5      CPY #$A5
B721-   FØ Ø3      BEQ $B726
B723-   38         SEC
B724-   BØ Ø6      BCS $B72C
B726-   A9 4D      LDA #$4D
B728-   8D AC 9A   STA $9AAC
B72B-   18         CLC
B72C-   A9 ØØ      LDA #$ØØ
```

```
B72E-  8D BF B3    STA $B3BF
B731-  A0 09       LDY #$09
B733-  B9 3D B7    LDA $B73D,Y
B736-  99 EB B7    STA $B7EB,Y
B739-  88          DEY
B73A-  10 F7       BPL $B733
B73C-  60          RTS
```

B73D.B750

```
B73D- 00 00 00
B740- 00 00 00 00 00 00 00 00
B748- 11 00 FB B7 BB B3 00 00
B750- 01
```

42 Save the code:

BSAVE WRITEPROTECT,A$B700,L$51

WRITEPROTECT uses the RWTS routine to read the VTOC from the disk into the VTOC buffer at $B3BB. In order to do this without disturbing BSW's use of RWTS, $B700—$B710 save the current RWTS parameters into temporary storage at $B73D and replace them with the VTOC-reading parameters from the table at $B747. Then the RWTS is called to read the VTOC.

If an error occurs, it may well be because the disk is unformatted so we exit as 'OK-to-write' by branching to $B72C. If no error occurs, the value at $B3BF is compared to our constant ($A5). If our protected disk was in the drive, this byte would match and the branch to $B726 would be taken. Otherwise, it is safe to write the disk so the carry is set and we branch to $B72C.

At $B726, $4D is stored at $9AAC (like the original), the carry is cleared indicating 'not-OK-to-write' and execution drops into the exit code at $B72C.

At $B72C, a 0 is stored at $B3BF to guarantee that a $A5 is never written out to a data disk since that would write-protect it. Then at $B731, the parameters saved at the start of the routine are restored before returning to the caller.

43 One other tiny piece of initializing code is needed to satisfy the BSW program's use of DOS. Enter it as follows:

BFC8: 20 DC AB A9 10 8D F0 B3
BFD0: A9 23 8D EF B3 60

and then:

BSAVE PATCH,A$BFC8,L$14

Your softkey slave disk is now ready to boot! We have used this softkey version of BSW to write this article and that has allowed us to weed out many of the bugs. We were able to test all three hardware-dependent versions to some degree, so you should have few, if any problems. The use of a fast DOS makes a welcome improvement. We have successfully used *Diversi-DOS* but have not tried any of the others, although they should work just as well.

The tutorial on the back of the BSW master disk is not protected and may be copied with the normal copy methods, but for maximum protection we recommend that you mark the disk as a ''master'' so BSW cannot write to it. This can be done by using Steps 1 to 4 of this softkey to INIT the disk on which you wish to copy the tutorial. Then, use the *DOS 3.3 FID* program to copy all the files from the original tutorial disk to the INITed back-up disk.

An alternative method is to use a disk-*ZAP* type program to modify the fifth byte of Track $11, Sector $Ø to an $A5. This method can be used to protect other disks that might be accidentally written by someone using the BSW program.

Sean Williams' APT for...

Castle Wolfenstein

How to Kill The S.S. Guards

If you are tired of running into S.S. guards, especially when you no longer have any grenades to destroy them with, use the following technique and any S.S. can be killed with the use of bullets.

1) As the S.S. guard approaches, point your gun at him and run into him.

2) As the screen is going through the collision routine press the key which makes the gun point at the S.S. guard again.

3) To take his bullet-proof jacket away, press U.

4) If the screen replies with *SEARCHING...*, run into the guard again. (Remember to point your gun at him again during the collision routine).

5) Press U again and the guard will lose his bullet-proof jacket.

6) The guard can now be killed with plain bullets.

Note: This procedure works best if you are wearing a vest yourself.

Canyon Climber

Datasoft

Softkey for Canyon Climber
by John Liska
(Hardcore COMPUTIST # 10, page 8)

Requirements:
Apple][with 48K
FID
One blank disk

Canyon Climber is an excellent 3-level arcade game in which you attempt to scale a hi-res version of the Grand Canyon.

To remove the copy-protection, you need to make some modifications to DOS so that errors generated by the nonstandard end of address and end of data marks are ignored. The single binary file can then be transferred to a normal 3.3 disk with the use of *FID*.

1 First boot up with a *DOS 3.3* disk and make the modifications to DOS:

```
PR#6
CALL -151
B925:18 60
B988:18 60
BE48:18
3D0G
```

2 BRUN FID and use the wildcard character (=) to transfer the one file on the original *Canyon Climber* disk to a standard 3.3 disk. You must do this because the file name contains embedded control characters.

```
BRUN FID
```

Once you have transferred the file, use a disk editor or *Copy][Plus* to change the file name so that its control characters are eliminated.

Caverns of Freitag

Muse Software

Backup For The Caverns Of Freitag
by C. J. Singer
(Hardcore COMPUTIST # 6, page 6)

Requirements:
Apple II, with 48K and Applesoft in ROM
One disk drive
FID from *3.3 System Master* disk
One blank disk
The *Caverns of Freitag* original disk

Being the parent of three kids who like to play games on the computer, my first thought after purchasing a program is how to back it up before the kids have an accident with the original.

What I usually first try to do is backup the disk with *Locksmith*, but I don't really care for this method because it doesn't allow you to look at the program to see how it works and it does not allow you to modify it.

Here's my first, albeit somewhat tedious, explanation of how to backup the *Caverns of Freitag*. I will provide another, easier method next. The game comes from Muse Software, of *Castle Wolfenstein* fame.

The Hard Way

1 Boot the *Caverns of Freitag* disk from slot 6:

PR#6

2 While the disk is booting, stop the *HELLO* program from running:

CTRL C

3 To make sure the *HELLO* program is in memory, type:

LIST

You should see *THE CAVERNS OF FREITAG* listed on your monitor.

If no program is listed, go back to Step 1 and try again.

4 *The Caverns of Freitag* uses a modified DOS, which has changed some of the DOS commands. The INIT command has not been changed, however, so we can INITialize a blank disk, which will contain the modified DOS along with the *HELLO* program currently in memory. So, insert a blank disk into your drive and type:

```
INIT HELLO
```

If you want to see how Muse has modified the normal DOS commands you can use a sector editing program, such as *ZAP* from *Bag of Tricks* or *DiskEdit*, to view Track $1, Sectors $7—$8. If you compare what you find with a list of the normal DOS commands you will find that:

the **SAVE** command has been changed to **LSDK**
the **CATALOG** is now **KSJFLKA**
the **MON** to **983**
and **BSAVE** to **87364**.

5 Now put your original disk back in the drive and use Muse's modified CATALOG (KSJFLKA) to see what files are on the disk:

```
KSJFLKA
```

6 Write down the list of files which are displayed so that you can save them to the disk you just initialized. You don't have to write down the *HELLO* file since it has already been transferred.

7 The first file on your list should be a binary file called *OILER* so load it into memory:

```
BLOAD OILER
```

8 When a new binary file is loaded into memory, its address is stored at $AA72—$AA73 (hi/lo format) and its length is stored at $AA6Ø—$AA61. So enter the Monitor and display the address and length of the file *OILER*:

```
CALL -151
AA72.AA73
AA6Ø.AA61
```

For the *OILER* file you should come out with an address of $Ø2DE and a length of $ØØD5.

9 Put your backup disk in the drive and use the 87364 (the modified BSAVE command) to save the file, using the address and length parameters you just determined in the previous step:

```
87364 OILER,A$Ø2DE,L$ØØD5
```

1Ø Repeat the process of BLOADing each binary file from the *Caverns of Freitag* original disk, determining its address and length and using the 87364 (BSAVE) command to transfer the files onto the backup disk.

11 For the Applesoft files on the list, you can just LOAD each one of them from the original disk and, then, LSDK (SAVE) them onto the backup disk.

Like I said; the above procedure is long and tedious, but it works. Now for the easy method!

The Easy Way

1 Go through Steps 1 through 4, outlined above.

2 Insert your *DOS 3.3 System Master* and boot it

```
PR#6
```

3 Load the *FID* program into memory at $68ØØ where it won't get overwritten when we boot the *Caverns of Freitag* disk in the next step:

```
BLOAD FID,A$68ØØ
```

4 Boot the original *Freitag disk* and, again, stop it from running the *HELLO* program:

```
PR#6
CTRL C
```

5 Enter the Monitor, memory move *FID* from $68ØØ back down to $8Ø3 and, then, run *FID* to copy the files:

```
CALL -151
8Ø3<68ØØ.9ØØØM
8Ø3G
```

6 Copy the files from the original disk to the blank, initialized disk using *FID*'s wildcard option.

Now that you have a backup copy you can look at the *Caverns of Freitag*, using CTRL C and **KSJFLKA** for CATALOG. You can LOAD or BLOAD to study or modify any of the files on the disk.

If you have a sector editor you may even want to change Track $1, Sectors $7—$8, so that they will contain the normal DOS commands, instead of the ones which Muse put on the disk.

Crush, Crumble & Chomp

Automated Simulations (Epyx)

Backing Up Crush, Crumble & Chomp
by Jeff Rivett
(Hardcore COMPUTIST # 7, page 5)

Requirements:
Apple][Plus or compatible
Replay II card and utility disk with *SOFTMOVE* program
Crush, Crumble, and Chomp
An initialized *DOS 3.3* disk with *HELLO* as the boot program
FID program from *3.3 System Master* disk

Crush, Crumble, and Chomp is copy-protected only on areas of the disk that contain the two main Applesoft programs. The rest of the files, and there are a few of them, can be transferred to a copy disk using any file transfer program.

The main (copy-protected) files are called:

```
C CTRL N CCMAIN
C CTRL N CC
```

1 Run *FID* (or any other file transfer program) and copy all the files, except the two copy-protected ones, from the original to the blank initialized disk:

BRUN FID

2 After transferring the unprotected files to your initialized copy disk, boot the original *Crush, Crumble and Chomp* disk:

PR#6

3 Make a *Replay* copy at the point where the set-up program starts and asks you the question *DO YOU WANT TO CONTINUE A SAVED GAME?*

4 Now, run *SOFTMOVE* from the *Replay II* utility disk. *SOFTMOVE* assumes that there was an Applesoft program in memory when you pushed the *Replay* button to make the copy. This

Applesoft program is put back into memory intact, with a normal DOS resident instead of the copy-protected DOS.

5 You can now save the Applesoft file to the disk that contains all the files you have transferred. Save it with the name C[CTRL N]CC:

SAVE C[CTRL N]CC

6 Boot the original again. This time let the set-up progress to the point where the play actually starts and then make another *Replay* copy. Repeat the *SOFTMOVE* procedure you used previously on the first protected program. Save this file as C[CTRL N]CCMAIN:

SAVE C[CTRL N]CCMAIN

That's all there is to it. These files can now be modified at will.

A last note: The *SOFTMOVE* program is a surprisingly useful utility. For instance, if you typed in a huge Applesoft program and forgot that there was no DOS in the machine, you might want to shoot yourself (but don't). With *Replay II* and *SOFTMOVE* you can make a *Replay* copy and have the file on disk in no time, almost as if DOS was in the machine all along.

Johnny Yukon's APT for...

Miner 2049er

Unlimited Bounty Bobs

For this APT you need to have an old F8 ROM on the motherboard or Integer card. This APT will let you choose the starting level and will give an unlimited number of Bounty Bobs to player number one.

1) Boot up Miner 2Ø49er and go through the joystick adjustment routine.

2) When the game asks *ONE OR TWO PLAYERS?*, hit [RESET] to get into the Monitor.

3) Enter the desired starting level minus 1 at $814 and the actual starting level at $812. For instance to start on level Ø5 type: **814:Ø4 N 812:Ø5**

4) Enter the following:

```
Ø972:A9 Ø3 8D 16 Ø8 8D 17 Ø8 4C 81 Ø9 N 981G
```

5) The game will start up and player number one will have an unlimited number of Bounty Bobs.

Data Factory 5.0

Micro Lab

Softkey For Data Factory Version 5.0
by L.S. Davis
(Hardcore COMPUTIST # 8, page 14)

Requirements:
48K Apple][Plus or equivalent
Two blank disks
DEMUFFIN PLUS (see **How To Make DEMUFFIN PLUS** article)

Like most of us, I try to follow the Golden Rule: back-up thy disks. I did just that with *Data Factory Ver. 5.0* using *Locksmith 4.1*. I followed the parms carefully, read the notes and yes, it works. However, to use the bit-copy of *The Data Factory* you must:

1 Boot the disk.

2 Open disk drive door and press RESET when the computer starts to beep repeatedly.

3 Close door and press RESET when the computer prints *BREAK IN 5*.

4 Use the program.

This procedure sounds simple and works well, but that irritating beeping noise causes wives to yell, dogs to howl and nerves to be on edge. What a pain. I just wanted a copy, not problems! Why did this disk act this way? Having read Hardcore COMPUTIST from issue no. I, I thought maybe the answer would be in there. So... I searched through my old issues.

Finding lots of good hints and ideas, I began. The first thing I saw when I booted the copy was *BREAK IN 5*. Applesoft, I hoped. Articles in Hardcore COMPUTIST had mentioned using *DEMUFFIN PLUS*, so I thought maybe I could *DEMUFFIN* the files over to normal DOS. I tried this, but had no luck. The program did try to work, but came up instead with the infamous *I/O ERROR*. Back to Hardcore COMPUTIST for more reading.

Finding several articles which referred to reducing error checks on copying, I began again. I looked through my ***What's Where In The Apple*** book and found an area of code that looked promising.

In a 48K Apple with *DOS 3.3* booted, the code at $B8DC—$B943 is a routine which reads in a sector of data from the disk. If any Read errors are encountered, the 6502's carry bit is set and the *I/O ERROR* message is printed. At $B942 is the instruction which sets the carry bit (SEC, op code $38), so if this instruction is changed from SEC to CLC (CLear Carry, op code $18) then any Read errors will be ignored. Hopefully, making this change would solve my problem.

I booted the *System Master* to clear the trash already in memory and made a **CALL -151** RETURN, entered **B942:18** RETURN (as if I knew what I was doing) and placed my disk with *DEMUFFIN PLUS* in the drive. I typed **BRUN DEMUFFIN PLUS**, then placed the copy of *The Data Factory* in the drive and (WOW!) no *I/O ERROR*s. It read the disk! Just luck? I wondered.

So, I turned off the machine and started over. I initialized two disks with just a *HELLO* program and made one copy of each *Data Factory* disk using the following steps:

1 Boot *System Master*:

```
PR#6
```

2 Clear the BASIC program in memory:

```
FP
```

3 Initialize two disks with *TDF* as the boot file:

```
INIT TDF
```

4 Enter the Monitor:

```
CALL -151
```

5 Change the instruction at $B942 from SEC ($38) to CLC ($18) so that any Read errors are ignored:

```
B942:18
```

6 Re-enter BASIC:

```
3D0G
```

7 Place a copy of *DEMUFFIN PLUS* in the drive and get up and running.

Note: If you don't have DEMUFFIN PLUS, you can create it by modifying MUFFIN. See the article on **How To Make DEMUFFIN PLUS** *in this volume.*

```
BRUN DEMUFFIN PLUS
```

8 There are two *Data Factory* disks: one *Report*, one *Utility*. DEMUFFIN all the files on each to a disk formatted in Step 2 above, using the 'wildcard' option (=). Replace the file called *TDF* on both disks.

9 When you have finished copying both disks, reboot with a normal 3.3 disk:

```
PR#6
```

Both of your *DEMUFFINed* disks are standard *3.3 DOS* and are COPYAable. However, on each disk is a file *TDF*. This file on each disk contains line **5** which reads: **PRINT CHR$ (4) "BRUN READER, A$8E00"**. This file contains the code which causes that obnoxious beeping. All we have to do is change the boot program (*TDF*) so that it does not load and execute this code.

10 Load the Applesoft file *TDF*, delete line 5 and then reSAVE TDF:

```
LOAD TDF
5
SAVE TDF
```

You should do this to both disks. If you want to, you can also DELETE the file *READER* which is not used after you delete line 5 in *TDF*.

In order to prevent a reboot if RESET is pressed, the binary file *AMPER FACTORY.OBJ0* needs to be modified.

11 Load the *AMPER FACTORY.OBJ0* file into memory (it loads at $8240):

```
BLOAD AMPER FACTORY.OBJ0
```

12 Enter the Monitor:

```
CALL -151
```

13 Change the instructions at $8340 from JMP $C600 (reboot the disk) to three NOP's:

```
8340: EA EA EA
```

14 Return to BASIC:

```
3D0G
```

15 Save the modified code:

```
SAVE AMPER FACTORY.OBJ0,A$8240,L$1305
```

The disk will now boot and work just like the original, except that you can now hit RESET without the disk rebooting.

I have run the deprotected copies and have not yet found any problems. I hope you have as much fun using this softkey as I had making it. You may notice that I worked from the copy of the original, not the original. This is much safer. That's another of the Golden Rules I like to follow: never use the original disk to make changes.

DB Master

Stoneware, Inc.

Softkey For DB Master
by Dan Lui
(Hardcore COMPUTIST # 7, page 6)

Requirements:
Apple][with 48K
One disk drive
DB Master (old version)
COPYA
The Inspector, or similar program
One blank disk

The old version of *DB (Data Base) Master* is protected very well. The disk contains three different protection schemes.

First, it uses half-tracks from $6.5—$22.5. Second, the closing address and data marks have been changed from the normal *DOS 3.3* $DE / $AA to $DF / $AA. Third, there is a nibble-count-like checking routine to check Track Ø.

Breaking the nibble-count scheme is the most difficult task of all.

The following procedure will unlock this program. It also works for DB's utilities disk:

1 Put in the *DOS 3.3 System Master* and type:

LOAD COPYA

2 Add the following lines to *COPYA*:

```
199 GOSUB 4ØØ
248 GOSUB 42Ø
259 GOSUB 42Ø
4ØØ POKE 47413,223: POKE 47423,171: POKE 475Ø5,223: POKE 47515,171
4Ø5 POKE 48351,2Ø1: POKE 48352,12: POKE 48353,1Ø5: POKE 48354,Ø:
    POKE 48355,24: POKE 48356,76: POKE 48357,1Ø7: POKE 48358,19Ø
41Ø POKE 48741,223: POKE 48742,188: RETURN
42Ø POKE 48741,1Ø7: POKE 48742,19Ø: POKE 47413,222: POKE 47423,17Ø:
    POKE 475Ø5,222: POKE 47515,17Ø
425 POKE 48741,1Ø7: POKE 48742,19Ø: RETURN
```

3 Save the new *COPYA* in case you run into errors.

SAVE DB COPYADB

4 Execute the program

RUN

5 After the disk has been copied, use the *Inspector*, or some other sector editor to read and modify the following sectors:

Track	Sector	Byte	From	To
00	03	$35	DF	DE
00	03	$3F	AB	AA
00	03	$91	DF	DE
00	03	$9B	AB	AA
00	0E	$0A	A2	D0
00	0E	$0B	00	12
01	0F	$C7	A9	60
03	01	$3E	20	60

6 Write-protect the disk before running.

The above procedure eliminates all three of the protection schemes Stoneware provided on the older version of the excellent *DB Master*.

Ferrel Wheeler's APT for...

Star Maze

Eliminating Snails and Wheels

It seems the pinwheel and the wall snail are extremely difficult to shoot so, instead of wasting a bomb on them, here's a method for destroying them and anything else in the game at no risk.

1) Drive repeatedly straight into a wall at a fairly slow speed until the ship is completely inside a wall.

2) Once inside, turn around and position the ship so that only enough of its nose is sticking out to still allow for firing.

When in this position, nothing can hit the ship, not even when the snail is walking on the same wall the ship is in!

You can even use this technique to go through walls, but this seems to use up a lot of fuel.

Essential Data Duplicator I

Utilico Microware

Copying the uncopyable - EDD
by Steven Zupp
(Hardcore COMPUTIST # 8, page 26)

Requirements:
48K Apple][or equivalent
One disk drive
Essential Data Duplicator Version I
A blank initialized disk with no *HELLO* program
A basic knowledge of machine language helps but is not necessary
Integer firmware card or other means of RESETing into the monitor

EDD (*Essential Data Duplicator*), in my opinion, is one of the most incredible copy programs around. My main reason for this praise is due to the fact that *EDD* requires few or no parameter changes to duplicate many copy-protected programs currently on the market. Its power almost makes *Locksmith 4.1* look like *COPYA*.

Unfortunately, along with its incredible copying abilities comes an incredible copy-protection. *EDD* uses four phases of super-fast track-arcing that sounds more like an army of cockroaches tap dancing than a disk booting. Instead of encrypting the data on the disk itself, the tracks of data are placed in a mosaic jumble of normal tracks, half tracks, and (yes, believe it or not) quarter tracks. This technique is loosely known as '**checkerboarding**' and is a very difficult scheme to crack.

Luckily for everybody, *EDD* is a one-time-only load program which leaves it open to attack from *snapshot*-type devices. In fact, a friend of mine armed with his *Wildcard* and some address changes successfully copied it, but his copy takes forever to load, is hard to modify, and takes up most of a disk. So, armed with my trusty Integer firmware card I decided to try it my way.

Getting The Data

This section deals with the breaking process in detail. If you just want to do it, then go to the section labeled '**Summary**.'

Boot *EDD*. Press any key a couple of times to get to the main menu, then flip the switch on the card to *Integer* (if it's not already there) and press RESET. You should end up in the Monitor.

Conveniently, *EDD* mainly uses the memory $800—$3FFF, with the hi-res picture residing at $4000—$5FFF. There is some code in the keyboard input buffer ($200—$300) but is has no effect on the main program, and I assume it is part of a disk-loading routine.

Before booting up our normal 3.3 slave disk we need to move the code at $800—$8FF to a safe location ($6000—$60FF) so it won't get stomped upon. To move the code out of the way type:

```
6000<800.8FFM
```

This moves $800—$8FF, to $6000—$60FF.

Now boot up *DOS 3.3* by inserting your initialized disk in the drive and typing:

```
C600G
```

This boots the disk in the drive in Slot 6. If your disk controller is in another slot, replace the 6 with that slot number; i.e. Slot 7 = C700G.

When the cursor appears, enter the Monitor by typing:

```
CALL -151
```

To relocate the portion of memory we moved back to its original location, type:

```
800<600.60FFM
```

Next we will make this hunk of data useable.

WARNING

You should now save this in case you mess up the steps following. If there is an accident, just BLOAD DATA and begin again at **Making It Useable**. To save it type:

```
BSAVE DATA, A$800,L$5800
```

Making It Useable

The main copy-protection routine in *EDD* starts at $21C9. Assuming you still have the *EDD* data in memory, and are still in the Monitor, make the following modification

```
21C9:60
```

This puts an RTS at the beginning of the routine that checks certain memory locations. All there is to be done now is to add a short routine which will perform some necessary housekeeping functions.

Since we will not be using hi-res page 2 ($4000), we can put our routine over it. Because the body of the program starts at $800, we must put a jump to our routine before that. So at $7FD we will put a jump instruction to $4000. Type this:

```
7FD:4C 00 40
```

In our routine there are three things that need to be done:

1. Disconnect the DOS pointers. The reason for this is *EDD* writes over DOS when in operation. When DOS is connected, the CSW (Character output SWitch) and KSW (Keyboard input SWitch) vectors are pointing to DOS. DOS looks at the characters coming in and going out and then sends them back along their merry way to the monitor COUT and KEYIN routines. If DOS is connected (the pointers are set for DOS locations) and those locations in DOS are erased, the computer will seemingly die, unable to input or output characters. If you want to learn more about this process refer to the section of this article titled **More On DOS Hooks**.
2. Change two Zero-Page locations. These are used by *EDD* to display which drives and slots are being used. These have nothing at all to do with the operation of the program, but it is nice to have them correct.
3. Jump to the actual start of the program (no, it's not $800).

Also changing the reset vector to C600 adds a nice touch so that when RESET is pressed, *EDD* will reboot instead of breaking into Applesoft. If your controller is in another slot you can change the slot (6) to whatever slot you want, although the wisest thing to do would be to move your controller to slot 6.

Examine the code below then type in the hex dump that follows.

```
4000-  A9 00      LDA #$00   ---------------------------------
4002-  8D F2 03   STA $03F2         Sets RESET vector
4005-  A9 C6      LDA #$C6              to reboot
4007-  8D F3 03   STA $03F3            from slot 6
400A-  20 6F FB   JSR $FB6F  ---------------------------------
400D-  A9 F0      LDA #$F0
400F-  85 36      STA $36
4011-  A9 FD      LDA #$FD
4013-  85 37      STA $37           Disconnects DOS
4015-  A9 1B      LDA #$1B
4017-  85 38      STA $38
4019-  A9 FD      LDA #$FD
401B-  85 39      STA $39    ---------------------------------
401D-  A9 60      LDA #$60
401F-  85 0A      STA $0A            Sets up screen
4021-  A9 60      LDA #$60         display locations
4023-  85 0B      STA $0B          and starts program
4025-  4C 5E 09   JMP $095E  ---------------------------------
```

```
4000: A9 00 8D F2 03 A9 C6 8D
4008: F3 03 20 6F FB A9 F0 85
4010: 36 A9 FD 85 37 A9 1B 85
4018: 38 A9 FD 85 39 A9 60 85
4020: 0A A9 60 85 0B 4C 5E 09
```

Now that you have entered the routine, it is time to save the whole thing. Type the following:

```
BSAVE EDD, A$7FD, L$382B
```

That's it! You can DELETE the file *DATA* now if you want to, or keep it on hand as a space taker-upper on your disk.

Just **BRUN EDD** whenever you want to use it, and hide the original in a locked safe in a bomb shelter and you should feel safer (watch out for safecracking rats wielding magnets, however.)

Summary

1 Boot *EDD* and press any key twice to get to the menu. Now use whatever device you have to enter the Monitor. After you are in the Monitor type:

```
6000<800.8FFM
```

2 Now insert your blank disk and type:

```
C600G
```

3 When the prompt appears, re-enter the Monitor:

```
CALL -151
```

4 Type in the new code.:

```
800<6000.60FFM
2C19: 60
7FD: 4C 00 40
4000: A9 00 8D F2 03 A9 C6 8D
4008: F3 03 20 6F FB A9 F0 85
4010: 36 A9 FD 85 37 A9 1B 85
4018: 38 A9 FD 85 39 A9 60 85
4020: 0A A9 60 85 0B 4C 5E 09
```

5 Save it as *EDD*:

```
BSAVE EDD, A$7FD, L$382B
```

That's all! Just type **BRUN EDD** to run it.

More On DOS Hooks

This exercise is for some rainy day when you have nothing better to do.

First boot a normal DOS disk. Whenever you get a cursor on the screen, type:

```
CALL -151
```

to get into the Monitor. Now we will check the present settings of the CSW and KSW switches. Type:

```
36.39
```

You should see:

```
36- BD 9E
38- 81 9E
```

This means CSW is set to $9EBD and KSW is set to $9E81, both DOS locations. Now we will erase DOS. Type:

```
8ØØ: ØØ
8Ø1<8ØØ.BFFFM
```

Wait about 5 seconds. Your Apple is temporarily dead. It will continue to be so until you press RESET to resurrect it.

When you press RESET, there will be a beep and you will see a status line indicating $9DC1. What happened is when you pressed RESET, the computer realized what happened, set the CSW and KSW switches back to normal, and showed you where the screw-up was.

When the move routine had finished, the Apple needed to output a character, the * prompt for the monitor. So it looked at the CSW pointer which still pointed at DOS, which we just erased. When it went to the location $9EBD all it found was zeroes. It couldn't signal an error. It couldn't even beep or show any sign of output or input. Well, fortunately we have the RESET key for such situations. The RESET key is wired directly to the Apple's microprocessor, which means it couldn't care less what the CSW and KSW pointers were. What happened is the RESET routine went through memory and fixed all the locations to default values, including our pointers. The Apple has a very good memory and it loves to beep at us when we screw up, and it does just that by beeping and telling us exactly where the error was.

Now that we're back on stable ground let's look at our pointers and see what's happened. Type:

```
36.39
```

You should see:

```
36- FØ FD
38- 1B FD
```

Note that these are the Monitor locations COUT1 and KEYIN. When DOS is not connected (like now) these are the values that are present. When DOS is connected all input and output goes to DOS and then to these locations.

Now clear the memory from $800—$BFFF again by typing:

```
800:00
801<800.BFFFM
```

After a few seconds you will see the Monitor prompt again. Why did the Apple die the first time we did this and not now? It is because the first time we did this the CSW and KSW pointers were pointing at the DOS locations which we later erased, causing massive heart failure. This time they were pointing at safe ROM routines which cannot be erased.

So remember to never erase or change the routines at $9EBD and $9E81 unless you know how, and have the incoming data ending up at $FDF0 and $FD1B.

Alternative devices

The *snapshot*-type devices currently on the market advertise the ability to stop the execution of any program. However, when I tried to use *Wildcard* to stop *EDD* and jump into the Monitor there were problems. Evidently, *EDD* can recognize that something's amiss when this is done because it changes parts of itself and will not resume operation. If you have another type of *snapshot* card, or want to try it anyway, just use it to break into the Monitor instead of a firmware board.

I will list what I know about *EDD* so that if you want to try fixing or avoiding this problem you can give it a try. My friend successfully made a *Wildcard* copy of *EDD* using a few address changes. This suprised me since I had found, when I had used it and gone into the Monitor, that it had changed itself. His copy took up a whole disk and was not easily modified. Here are the changes he made if you want to try them with your *snapshot*er or figure out their meaning.

```
113A: A9
113B: 0B
113C: EA
21D8: 00
21DF: 00
21DE: 00
```

Some of the area around $CE9 was also changed, and there are probably others.

Essential Data Duplicator III

Utilico Microware

Copying Essential Data Duplicator III
by Joseph Leathlean
(Hardcore COMPUTIST # 10, page 7)

Requirements:
48K Apple][or Apple][Plus
One disk drive
Blank disk with no *HELLO* Program
Integer Firmware Card or other means to RESET into the Monitor

The method of backing up the *Essential Data Duplicator I* which appeared in the previous *EDD* softkey can be modified so that the technique also works on *EDD III*. Only two changes need to be made.

On *EDD III* the main copy-protection scheme starts at $3EE8 rather than at $21C9. This byte has to be changed to $60 (RTS).

The program's main entry point has also been moved from $095E. On the version dated January 25, 1984, the new entry point is $0955. On later versions the entry point is $0953. You may have to experiment with your version to find the proper entry point.

So if you want to perform this *EDD III* softkey, follow the procedure explained in the previous *EDD* and make the following changes:

1 Place a $60 at address $3EE8 instead of at $21C9.

2 For the January 20, 1984 release of *EDD* change address $4026 in the hexdump to a $55. For later versions try changing it to a $53.

Gold Rush

Sentient Software

Deprotecting Gold Rush
by Clay Harrell
(Hardcore COMPUTIST # 9, page 7)

Requirements:
48K Apple][with at least one *DOS 3.3* disk drive
Old-style F8 Monitor ROM or Super-Saver ROM
Blank disk
Gold Rush

It seems that there are many 'maze' games flooding the Apple market, now. Most have pretty much the same theme of eating all the dots before you get destroyed. I do enjoy these games, but variety is the spice of life! *Gold Rush* uses the 'avoid the bad guys' theme without the maze. It is entertaining, fun, and different from other chase 'em games. Being intrigued by the game, I wanted to learn more about it...

Gold Rush boots fairly quickly and is a 'single-load' game, which means it does not require any other data from the disk after the initial load. These types of programs are candidates for BLOADable files.

When booting the disk, an Applesoft prompt appears at the bottom left of the screen which means there is some sort of modified DOS present. Disks like these are excellent candidates for deprotection with Super IOB. The Swap controller is probably the most versatile since one doesn't usually have to figure out the protection used when using it. With this in mind, we merely boot *Gold Rush*, RESET into the Monitor and move RWTS from $B8ØØ—$BFFF down to $19ØØ where the standard Super IOB swap operates. This is done from Monitor with the command **19ØØ<B8ØØ.BFFFM**

Now boot a 48K slave disk and run *SUPER IOB.SWAP* which is Super IOB with the NewSwap controller merged (see **Super IOB** in this volume). Remember that booting a slave disk only destroys memory from $ØØ—$8FFand $96ØØ—$BFFF, so this keeps our *Gold Rush* RWTS safe at $19ØØ—$2ØFF.

After making the copy, our new disk is quite CATALOGable. Also, the disk is now deprotected and completely *COPYA*able and runs fine (providing the boot file name is *BOOT* instead of the normal *HELLO*). But remember, this is a single-load game, and as I mentioned before, a good candidate for a BLOADable file.

```
CATALOG
```

reveals one file named *BOOT*. You can BRUN this file and the game will load and execute just fine. But we want this game in a file that we can *FID* to another disk or BLOAD. To start the analysis:

```
BLOAD BOOT
```

Now enter the Monitor with:

```
CALL -151
```

and type:

```
AA60.AA73
```

The first two bytes listed are the length of the last BLOADed file, and the last two bytes listed are the address the last BLOADed file was loaded at. These are in 'bassackward' order, with the low byte first and the high byte last (in other words, 00 03 means $0300, or 88 45 means $4588). We can see from this listing that the file *BOOT* loads at $300 and is fairly short (less than $100 bytes long).

If you examine the code at $300, you will find that Sentient is using second stage DOS to load in the game from $800 to $95FF, and then jumps to $B00 to start the game.

Sentient Software RWTS is not as friendly as the normal RWTS. They load $1B with the track number, $1C with the sector number and, $1F with the top page to load to. Then they Jump SubRoutine to $B7B5 to do the load. Look at the code and try to understand it (If you cannot, well, forget it).

After some groaning and grunting, we now know that *Gold Rush* loads from $800—$95FF and starts at $B00. With this necessary information in hand, here are the exact steps for deprotecting *Gold Rush*:

1 Boot *DOS 3.3* and INITialize a disk with a null *HELLO* file.

```
FP
INIT HELLO
```

2 Boot *Gold Rush* and after the game loads and the drive stops spinning, RESET into the Monitor.

3 Move page eight out of the way so we can boot:

```
6400<800.8FFM
```

4 Boot the disk you just initialized:

```
C600G
```

5 Enter the Monitor:

```
CALL -151
```

6 Move $800 to its original location:

```
800<6400.64FFM
```

7 Alter the program so that it JuMPs to the entry point:

```
7FD:4C 00 0B
```

8 Tell DOS that we can BSAVE more than 121 sectors:

```
A964:FF
```

9 Save *Gold Rush*:

```
BSAVE GOLD RUSH,A$7FD,L$8E04
```

Gold Rush is now deprotected into a BRUNable file that you can *FID* to any disk.

APT for Proving Grounds Of The Mad Overlord...

Wizardry

Creating a powerful Bishop

It seems as though a lot of Wizardry players already know how to create a very powerful bishop, but here is the technique anyway:

1) Take the bishop into the maze and then camp.

2) Inspect the bishop's character and choose the identity I option.

3) Keep trying to identify item #9 until the word *SUCCESS* appears at the bottom of the screen.

The bishop should now have 100,000,000 experience points.

Krell LOGO

Krell Software Corp.

Using Super IOB To Copy Krell LOGO
by Andrew Harrison
(Hardcore COMPUTIST # 10, page 8)

Requirements:
Apple][plus with 64K or //e
Krell *LOGO*
Super IOB
Swap controller
One blank disk
Integer Firmware card or other means of entering the Monitor

Krell *LOGO version A* can be copied with the use of the Super IOB program which has the Swap controller installed. Super IOB and the Swap controller can both be found in the **Super IOB** article in this volume. You will also need some way to RESET into the monitor so that the *LOGO* RWTS can be saved to disk.

1 Boot up with the original Krell *LOGO* disk and then RESET into the Monitor.

2 Move the RWTS to a safe location:

```
1900<B800.BFFFM
```

3 Boot up a *DOS 3.3* slave disk with no *HELLO* program:

```
PR#6
```

4 Save the Krell *LOGO* RWTS to the same disk that has the Super IOB program on it:

```
BSAVE LOGORWTS,A$1900,L$800
```

5 Format a blank disk with the *HELLO* program listed below:

```
FP
10 PRINT CHR$(4)"BLOAD BANNER.SHP"
```

```
20 PRINT CHR$(4)"RUN KRELL.START"
INIT HELLO.LOGO
```

6 Load the *Super IOB* and type in or EXEC in the Swap controller:

7 Change line 10010 to read:

```
10010 PRINT CHR$(4)"BLOAD LOGORWTS,A$1900"
```

8 Run the *Super IOB* program copying from the original Krell *LOGO* disk to the disk initialized in Step 5. **Do not** reformat the duplicate disk.

```
  RUN
```

9 After the copy has been made, **RENAME** the old *HELLO* file on the copied disk as *LOGO.START*

```
RENAME HELLO,LOGO.START
```

10 The file that is now called *LOGO.START* has a CALL 2167 which needs to be removed in order for the copy to work properly. LOAD the *LOGO.STARTUP* file and remove this CALL. It should be in the first line of the program. Be sure to reSAVE the program after you have made the change.

Eric Holman Whitaker's APT for...

Castle Wolfenstein

The S.S. will not follow you if...

Hold up an SS Stormtrooper and search him. After you have confiscated his bullets (search him twice to be sure), you can leave the room without fear of the SS following you.

Castle Wolfenstein tries to ge the S.S. to follow you and then shoot you on the run.

If the S.S. gent is out of bullets, the game decides that he is not much good and does not send him trailing after you.

Legacy of Llylgamyn

Sir-Tech Software, Inc.

Breaking Windows: Softkey For Legacy of Llylgamyn
by Jim Kaiser
(Hardcore COMPUTIST # 8, page 1Ø)

Requirements:
Apple][, Apple][plus, Apple //e or compatible with one disk drive
A Sector Editing Program, such as *DiskEdit*
COPYA from *DOS 3.3 System Master* Disk
Two blank disks

Legacy of Llylgamyn is one of the scenarios in the *Wizardry* Series. After trying to back up my original using the softkey presented in Hardcore COMPUTIST # 4 and not succeeding, I set out to do it myself. The program uses what Sir-Tech calls 'Window Wizardry', a technique they probably implemented upon seeing a LISA computer in action. I liked the 'window' concept so much that I purchased *Legacy of Llylgamyn* as soon as it came out.

Both sides of *Legacy of Llylgamyn* can be copied with *COPYA* if DOS is patched so that any errors encountered when reading or writing are ignored. The *COPYA* version of the boot side (side B) needs a sector-edit performed on it so that the routine to check for a non-original disk is circumvented. This routine is on Track $1A, Sector $ØC of the boot side. The scenario disk (side A) is unprotected and the *COPYA* version of it will work just fine.

1 Get out your *DOS 3.3 Sytem Master* disk and run *COPYA*:

RUN COPYA

2 After the drive has stopped, halt the program:

CTRL C

3 Prevent *COPYA* from reloading *COPY.OBJØ* by deleting line 7Ø:

7Ø

4 Enter the Monitor:

CALL-151

5 Patch DOS so that any Read or Write errors are ignored:

```
B7CØ:18
```

6 Return to Applesoft:

```
3DØG
```

7 Run the program:

```
RUN
```

8 Copy both sides of the original *Legacy of Llylgamyn* disk.

9 After you have copied both sides, put the original disk away in a safe place.

1Ø Run your sector-editing program on the backup copy.

11 Read Track $1A, Sector $ØC of the boot disk (side B)

12 Start editing at byte $15, entering the following bytes:

```
DØ 16 EA AD 2D ØØ CE FB ØØ DØ F8
AD DE ØØ A9 Ø1 48 A5 Ø1 48 A5 ØØ
48 6Ø A9 ØØ FØ ED
```

13 Write the sector back to the disk.

The scenario side (side A) is not protected on *The Legacy of Llylgamyn*, so no modifications to the scenario side of the disk are needed. You should now have a working copy and, by the way, you have to write-protect both sides of this disk. The only difference between the master scenario and the duplicate is that the master is write-protected. This disk is now *COPYA*able

OF SPECIAL INTEREST

I was bothered and didn't like to always have to switch disks for the scenario, just to enter the master disk, and then the duplicate. So, I set out to find how I could modify the disk so that I could enter the duplicate without a write-protect tab over the hole and still be able to have the program not give the dreaded *NOT A MASTER DISK* error. To make this modification, you should edit Track $6, Sector $A, Byte $73, Change it from a CB to a C3. You no longer have to put in the master and then, the duplicate. Just enter the duplicate and press RETURN twice. Now, if I could just figure out how to pronounce ''Llylgamyn''.

Mask Of The Sun

Ultrasoft, Inc.

Softkey For Mask Of The Sun
by John J. Liska
(Hardcore COMPUTIST # 7, page 27)

Requirements:
Apple][Plus
One disk drive (two are preferred)
Mask of the Sun program disk
DOS 3.3 System Master
One blank disk

Now you can unlock one of the most maddening adventures I have ever attempted to solve.

In this diabolical foray, your quest is to retrieve the legendary 'Mask of the Sun' and live to tell the tale. You will encounter many things along the way which will defy common sense and your own intuitive logic, yet must be met.

The basic procedure is the same as the modified *COPYA / DOS* combination used to unlock *Zork I* in the Book Of Softkeys Volume I. The only change to DOS seems to be the use of non-standard end marks on Tracks $03—$22 and incorrect checksums on the DOS tracks.

The first problem is to get the information off the original disk and onto the copy disk. Here is how:

1 Boot the *DOS 3.3 Master* Disk:

```
PR#6
```

2 Clear program in memory and initialize the front side of the disk you wish to copy to:

```
FP
INIT SIDE A
```

3 Drop into the Monitor:

```
CALL -151
```

4 Modify DOS so that it ignores end marks and checksums.:

```
B92D:18 60
B989:18 60
```

5 Put back in the *System Master* and use *COPYA* to copy SIDE B to the back side (the side you **didn't** initialize in step 2):

```
RUN COPYA
```

6 When finished with the back side, use *FID* to copy every file (using the wildcard (=) and prompting features of *FID*) from side A except *LL(V27)* to the front side of the freshly INITialized disk:

```
BRUN FID
```

We now have all the files copied except the one that checked for copy-protection. In order to make this disk work, we have to make a start-up file.

7 Type in this start-up program and save it to the front side of the copy:

```
10  PRINT CHR$( 4) "EXEC DISK"
SAVE SIDE A
```

That's it! Whenever you boot, you will get a *FILE NOT FOUND* message, but that is O.K. The reason for this is because the text file named *DISK* is trying to *BRUN LL(V27)*, but since it isn't on this disk, *DISK* just goes on to the next command.

You now have a copy of *Mask of the Sun* on a normal disk for you to examine until you bust (or solve the adventure, whichever comes first).

Now, would someone please tell me how I can get past the BLASTED snake!

Paul Andersen's APT for...

Serpentine

Extra Snakes

After all the snakes have appeared in the cage, type ! $ to get extra snakes.

Minit Man

Penguin Software

Backing Up Minit Man
by Clay Harrell
(Hardcore COMPUTIST # 1Ø, page 28)

Requirements:
48K Apple with old-style (or modified) F8 Monitor ROM
One disk drive with *DOS 3.3*
Two blank disks
Minit Man
Super IOB

Minit Man is a *Choplifter*-type game from Penguin Software. The game plays well and has nice graphics, demos and title pages. Unfortunately, its slow loading is practically unbearable at times. My original intent was to make *Minit Man* load at the speed of light... But in order to do this I had to first remove the disk's copy-protection. *Minit Man* proved to be a challenge on which I had to use some of my own programming to overcome Penguin's protection. First, let's talk about the protection used in *Minit Man*.

You can protect a program by various means, or you can protect a disk full of programs with some sort of DOS modification. DOS modifications are the least successful of protection schemes, since someone always seems to find a way to copy all the files onto a normal DOS disk, eluding all the protection.

MUFFIN—DEMUFFIN—DEMUFFIN PLUS

The classic program for dealing with modified DOSes is *DEMUFFIN PLUS*. It works much the same way as Apple's *MUFFIN* program works.

MUFFIN was written to read files from a *DOS 3.2* disk and then write them to a *DOS 3.3* disk.

DEMUFFIN was a variation of *MUFFIN*, allowing the hardcore *DOS 3.2* user to copy disks from *DOS 3.3* to *DOS 3.2*.

DEMUFFIN PLUS operates on the same principle, but uses whatever DOS is in memory to read the disk, and then writes out to an initialized *DOS 3.3* disk. While this is a powerful utility, it only works with programs that are based on DOS and that have a CATALOG track with normal, or somewhat normal, files.

Now, with this tidbit of information in mind, how do we know that a disk uses a modified DOS? There are many hints, the foremost being the appearance of a BASIC prompt at the bottom of the screen during booting.

Some publishers have bypassed the routine that outputs the prompt, but you can still guess that there's a modified DOS present if the boot sounds like a normal DOS boot, but the disk won't copy with *COPYA*.

Modified Minit DOS

Minit Man's protection scheme falls into this category of modified DOS protection. Upon booting the disk, the normal DOS boot sounds are heard and a prompt appears at the bottom left of the screen. Instead of attempting to use *DEMUFFIN PLUS*, we are going to use a different approach to the deprotection of *Minit Man*, using Super IOB.

The actual modification on the *Minit Man* disk is that the end of address and end of data markers have been changed on every track to DA AA instead of the usual DE AA. In addition, the start of address markers on every other track have been changed to D4 AA 96 instead of D5 AA 96.

The Super IOB controller starts copying from Track $2 (we do not want the protected DOS residing on Tracks $Ø—$1) of the protected disk and writes back to the same tracks of our normal *DOS 3.3* Disk.

With the Super IOB controller in hand we can now start to deprotect *Minit Man*. Here are the first few steps:

1 Type in the *Minit Man controller* (listed at the end of this softkey) and save it with the *SAVE CONTROLLER* program (see **Controller Saver** article in this Book Of Softkeys) or just **LOAD SUPER IOB** and type in the *Minit Man controller* and go to Step 2.

```
EXEC SAVE CONTROLLER
```

2 Copy *Minit Man* with its controller placed into Super IOB

3 Boot a normal *DOS 3.3* Disk (preferably with a fast DOS since the load is so slow on the original *Minit Man*) and initialize the disk with *HELLO* as the boot program. Then *FID* all the files to the freshly INITed disk.

```
INIT HELLO
BRUN FID
```

When the copy is done you will have the bulk of *Minit Man* deprotected. Now we must make the programs on the *Minit Man* disk compatible with *DOS 3.3*, since the protected DOS has some built-in routines that normal DOS does not. (Notice you can now CATALOG the disk and see the files that make up *Minit Man*).

4 Clear any program in memory and get the *HELLO* program:

```
FP
LOAD HELLO
```

5 Make the following changes:

```
1 HOME
5 HGR2: HGR
6 CALL 33072
```

6 Save the program:

```
SAVE HELLO
```

Now some other files on the disk need to be changed to complete the deprotection. Follow these steps:

7 Remove the file *PLAYGAME PROG*:

```
DELETE PLAYGAME PROG
```

8 Clear any program in memory and type this short one:

```
FP
1 PRINT CHR$( 4); “EXEC PROG”
```

9 Save it:

```
SAVE PLAYGAME PROG
```

10 Replace the program *DEMOSETPROG* with the following:

```
DELETE DEMOSETPROG
FP
1 PRINT CHR$( 4); “EXEC DEMO”
SAVE DEMOSETPROG
```

Now you must create some text files to load in the game and/or demo files. You may use a word processor that creates **normal** text files or you may create an Applesoft program to create them. ***Note: The file names must be typed exactly as shown or you will get a ?FILE NOT FOUND error.***

11 Create the text file *PROG* that contains:

```
BLOAD DP.7.2.SHAPES
BLOAD DPBC,A8516
BLOAD DP7,A16384
BLOAD PACPICS,A2048
BLOAD DP.7.2.ANNEX,A22574
BLOAD MEMSORT5C01,A$5C01
```

```
LOAD PACMOVE,A$6ØØØ
BLOAD TRAINS,A14ØØØ
BLOAD DP.7.2.ANMX
CALL 23553
```

12 Now create the text file *DEMO* that contains:

```
BLOAD DP7.DEMO,A16384
BLOAD DPBC.DEMO,A8516
BLOAD DP.7.2.SHAPES.DEMO
BLOAD PACPICS.DEMO,A2Ø48
BLOAD DP.7.2.ANNEX.DEMO,A22574
BLOAD MEMSORT5CØ1.DEMO,A$5CØ1
BLOAD PACMOVE.DEMO,A$6ØØØ
BLOAD TRAIN.DEMO,A14ØØØ
BLOAD DP.7.2.ANMX.DEMO
CALL 23553
```

Be sure to carefully enter the text files exactly as shown above. Have fun with lightning fast *Minit Man*!

Minit Man controller

```
1ØØØ REM MINIT MAN CONTROLLER
1Ø1Ø TK = 2 :ST = Ø :LT = 35 :CD = WR
1Ø2Ø T1 = TK : GOSUB 49Ø : GOSUB 111Ø
1Ø3Ø GOSUB 43Ø : GOSUB 1ØØ :ST = ST + 1 : IF ST < DOS THEN 1Ø3Ø
1Ø4Ø IF BF THEN 1Ø6Ø
1Ø5Ø ST = Ø :TK = TK + 1 : GOSUB 111Ø : IF TK < LT THEN 1Ø3Ø
1Ø6Ø POKE 475Ø5 ,222 : POKE 47413 ,222 : GOSUB 23Ø : GOSUB 49Ø :TK = T1
     :ST = Ø
1Ø7Ø GOSUB 43Ø : GOSUB 1ØØ :ST = ST + 1 : IF ST < DOS THEN 1Ø7Ø
1Ø8Ø ST = Ø :TK = TK + 1 : IF BF = Ø AND TK < LT THEN 1Ø7Ø
1Ø9Ø IF TK < LT THEN 1Ø2Ø
11ØØ HOME : PRINT : PRINT "DONE^WITH^COPY" : END
111Ø POKE 475Ø5 ,218 : POKE 47413 ,218 : IF TK / 2 = INT (TK / 2 ) THEN 23Ø
112Ø RESTORE : GOTO 19Ø
63Ø1Ø DATA 212 ,17Ø ,15Ø
```

Mouskattack

On Line Systems

Backing Up Mouskattack
by Clay Harrell
(Hardcore COMPUTIST # 7, page 6)

Requirements:
48K Apple
One disk drive, with *DOS 3.3*
DOS 3.3 System Master
Mouskattack
One blank, initialized 48K slave disk

Although *Mouskattack* is a rather old and not so thrilling maze game, it warrants discussion on deprotection methods and the use of DOS from protected programs.

Upon booting the *Mouskattack* disk, the prompt appears on the lower left side of the screen, indicating that a somewhat normal DOS is used by the program.

If you boot a normal *DOS 3.3* disk, then put your *Mouskattack* disk in the drive and type CATALOG, a directory does appear. You will not see any files, just copyright notices and names of the authors involved. These, however, are files in the directory.

The disk seems almost unprotected and to confirm this, I made a copy with *COPYA* from the *DOS 3.3 System Master*. It reads the non-DOS tracks slowly. This is due to the sector skewing used by On-Line in hopes of a faster loading game and not due to the protection. However, the people at On-Line were not too successful in carrying out their intention and we shall see why, in a moment.

The first three tracks (the DOS tracks) read at the normal speed because they are normal DOS, just like those in your *DOS 3.3 System Master*.

After making my *COPYA* copy, I used a Disk Editor to read in Track $11, Sector $F, of the disk. This is the first sector of the CATALOG track.

On-Line was able to make the directory appear without file types and sector lengths using a simple technique. The first seven (or six) characters in the directory name are back spaces. Starting with Byte $ØE, change the first seven characters to anything but control characters, numbers or spaces, and you can load and examine the files like any other *DOS 3.3* files.

Don't bother with the other file names, though. They are all blank files. The only one we are interested in is the first directory entry, *MOUSKATTACK*.

Snooping Through

Now, get back into BASIC and CATALOG the copy of *Mouskattack*. The first entry of your catalog should have a binary file, four sectors in length, called
CTRL H CTRL H CTRL H CTRL H CTRL H CTRL H CTRL H MOUSKATTACK.

You can now BLOAD this file and snoop through it.

At this, point you might ask how I knew to do this. Since *Mouskattack* has a normal DOS on it, by loading in Track $1, Sector $9, with a Sector Editor, you can see which file is the boot file or *HELLO* program.

Sure enough, *MOUSKATTACK* is preceeded by seven CTRL H s (i.e., backspaces). Therefore, you know that *MOUSKATTACK* is the first file you should snoop through.

BLOAD the file CTRL H CTRL H CTRL H CTRL H CTRL H CTRL H CTRL H *MOUSKATTACK*, enter the Monitor and examine locations $AA72—$AA73. This will tell you the loading location of the last BLOADed file. It will appear 'bassackwards' with the low byte first and the high byte second (i.e., ØØ Ø8 is equivalent to $Ø8ØØ). Do a **8ØØL** to list the first screen-full of the program.

Upon examination, you see that the accumulator is loaded and then stored in a location within DOS. After this happens a few times, you will see a JSR (jump to subroutine) at $B7B5. Now you may ask:

What's all this stuff?

On-Line is using the second stage of DOS to load in the game of *Mouskattack*. You are looking at the first stage (of three stages) of the load and are actually tracing the boot as described in earlier issues of Hardcore COMPUTIST, but using normal DOS.

The listing will appear as follows:

```
Ø8ØØ-     LDA #$ØØ
Ø8Ø2-     STA $B7EB  ;Volume Ø matches anything
Ø8Ø5-     LDA #$Ø1
Ø8Ø7-     STA $B7F4  ;Command code, 1 = Read
0'8ØA-    LDA #$18
Ø8ØC-     STA $B7EC  ;Track # to start loading from
Ø8ØF-     LDA #$Ø3
```

```
Ø811-        STA $B7ED  ;Sector # to start loading from
Ø814-        LDA #$95
Ø816-        STA $B7F1  ;High byte of page to load to
Ø819-        LDA #$ØØ
Ø81B-        STA $B7FØ  ;Low byte of page to load to
Ø81E-        LDA #$Ø3
Ø822-        STA $14    ;Sector counter in zero page
Ø824-        LDA #$B7
Ø826-        JSR $B7B5  ;RWTS sector read routine
Ø829-        BCS $Ø822  ;Branch to $822 if Error
Ø82C-        INC $B7F1  ;Increment page to load
Ø82F-        INC $B7ED  ;Increment sector to load
Ø831-        DEC $14    ;Decrement sector counter
Ø833-        BNE $Ø822  ;If sector # <> Ø branch to $822
Ø835-        JMP $95ØØ  ;Jump to $95ØØ
```

Starting with Track $18, Sector $3, the data is being loaded into location $95ØØ.

The sector number and memory page in which the data will be stored is incremented and the process continues until Sector $F of Track $18 is reached. Jump to location $95ØØ to start the next stage of the load. If you jump to the monitor at location $835 instead of $95ØØ, you can examine the next stage of the load. Do this by typing:

```
835:4C 59 FF N 8ØØG
```

At $95ØØ, the same thing happens again but in a larger perspective. The codes become more obscure and difficult to follow, so I won't list them here.

But now you ask:

What was the purpose of this exercise?

By examining the code at $95ØØ, you can find the starting page at which *Mouskattack* loads. You can even find the starting location! Can you see where?

The starting location of the loading program is at $AØØ and the starting location of the loaded program is at $53ØØ. This is the reward for all your labor!

Since the load code lives at $95ØØ and DOS occupies $9DØØ and up, *Mouskattack* must live at $AØØ—$94FF, with a starting location at $53ØØ. Not mentioned is the third load which overwrites the demo code. This load is the only part of the disk that is really protected. Our *COPYA* copy will work up to this load, but not past it, for the game.

Keep in mind that the code at $9ØØ—$95FF is not destroyed by a slave boot. Since *Mouskattack* lives within this area, it then becomes easy to crack.

Here is the procedure:

1 Boot your original copy of *Mouskattack*.

2 After the demo and, when the prompt *HOW MANY PLAYERS?* appears, insert a blank, initialized slave disk.

3 Hit RESET. It does not matter if you have an old-style Monitor or not. With an Auto-Start Monitor, your slave disk will boot.

4 Enter the Monitor:

```
CALL-151
```

5 To enable you to save larger binary files to disk, type:

```
A964:FF
```

6 Type in:

```
9FD:4C 00 53
```

7 Save the program:

```
BSAVE MOUSKATTACK,A$9FD,L$8B03
```

8 Have fun with the game!

Remember how I mentioned that the load of *Mouskattack* was rather slow? This is due to the method in which sectors are read in from the disk.

Also, the sector number increments along with the page number loaded to. If On-Line had started with the high page number, instead of the low page number and read the sectors in decreasing order, instead of incrementing order, the load would have taken a fourth of the time, providing the manufacturers had used normal *DOS 3.3* skewing.

This is the logic that the fast-loading DOSs have taken in order to increase loading speed and save time.

Music Construction Set

Electronic Arts

Softkey For Music Construction Set
by Jim Waterman
(Hardcore COMPUTIST # 9, page 7)

Requirements:
Apple][Plus, or compatible
Music Construction Set
Bit-copy program such as *Locksmith 4.1* or *Copy II Plus ver 4.1*
One blank disk

1 Use your bit-copier to copy Tracks $0—$22 from the *Music Construction Set* onto a blank disk.

2 Boot a normal *DOS 3.3* disk.

3 Insert the copy of *Music Construction Set* and load the file called *A4*:

```
LOAD A4
```

4 Enter the Monitor and change the bytes at $913A—$913B to NOP's:

```
CALL -151
913A:EA EA
```

5 Save *A4* with the changes you have made

```
BSAVE A4,A$4A00,L4B60
```

6 The *Music Construction Set* may now be copied with any normal copy program such as *COPYA*.

Pandora's Box

Datamost, Inc.

Deprotecting Pandora's Box
by Clay Harrell
(Hardcore COMPUTIST # 6, page 5)

Requirements:
Apple, with 48K
One disk drive, with *DOS 3.3*
A sector editor, such as *The Inspector* or *Disk Zap*
A blank disk
Pandora's Box

When program publishers buy a protection scheme, they generally use it for as many programs as possible to get the most for their money (yes, protection schemes are just programs that people write and sell). The advantage of this is that once you learn what they are doing, it is easy to backup many of their programs.

Datamost and Infocom are two examples of this. If you can backup *Zork I*, then you can backup *Zork II* and *Zork III* and *Deadline* and the rest.

The people at Datamost used a modified DOS for many of their programs until about April 1983. After that they started using a different scheme of protection on a lot of their programs. But the company published at least seven good games before April 1983 and *Pandora's Box* was one of them.

The deprotection method I am about to describe will apply to many of them, but we will be using *Pandora's Box* as an example.

Modified DOS

As we said, Datamost uses a modified DOS for its protection scheme. Normally, this is apparent from the BASIC prompt that appears on the screen after a few seconds into the boot.

To hide this,Datamost turns on the hi-res screen right when the boot starts. But we still know there is a modified DOS because of the way the boot sounds. Listen to your normal DOS disks boot.

You will hear the same sound every time. First, the drive spins for half-a-second or so. This is Track $Ø, Sectors $Ø—$9 getting loaded into $B6ØØ—$BFFF.

Then, you hear the drive's read-write head slide up to track $2 to load in the rest of DOS. Tracks $2—$1 are read in quickly and the read-write head slides up to Track $17, and the *HELLO* program is located and run.

Now, listen to *Pandora's Box* load in. You will hear the same sounds. This is a dead giveaway that a modified DOS is being used.

What to do

Whenever a modified DOS is used the first thing you should do is boot a normal DOS disk and defeat the DOS error-checking. DOS checks the carry bit to determine if any errors have occurred in a disk access.

If the carry bit is clear, DOS assumes that everything is OK and just keeps on going. The routine that gets jumped to if an error is suspected is at $B942. This simply sets the carry bit and returns to the calling routine.

To defeat the error-checking, we only have to change $B942 to $18, instead of $38. This simple modification will allow us to copy previously uncopyable disks with *COPYA*.

All that is left to do is to change the Datamost DOS just slightly so that it will live in a normal *DOS 3.3* environment. At Track $Ø, Sector $3, change Byte $91 from $DF to $DE. What the manufacturers have done to make their disk 'uncopyable' is to change the epilogue byte from the normal $DE to $DF. This will sufficiently confuse the copy program, preventing easy copies. (If you do not know what is meant by an 'epilogue byte', please refer to the **Beneath Apple DOS** book by Don Worth and Pieter Lechner. This manual is indispensible for further understanding of DOS).

In addition, Byte $42 should be changed from $38 to $18 on the same track and sector. This is an insurance policy, more or less, that everything will work correctly in the normal *DOS 3.3* environment.

What you are actually doing is changing Byte $B942 in DOS, as we did before to make the *COPYA* copy, but you are doing it directly to the disk for permanence.

These two modifications are all we need to do to make *Pandora's Box* a *COPYA*able disk.

The Steps

In step-by-step fashion, here's what you should do to make *Pandora's Box COPYA*able:

1 Boot normal *DOS 3.3*

2 Enter the Monitor:

```
CALL-151
```

3 Change byte $B942 from $38 to $18 by typing:

```
B942:18
```

4 Execute the *COPYA* program:

```
RUN COPYA
```

5 Copy *Pandora's Box* to a blank disk.

6 Re-boot normal *DOS 3.3* and run your sector editor. Change the following bytes:

Track	Sector	Byte	From	To
ØØ	Ø3	42	38	18
ØØ	Ø3	91	DF	DE

7 Write the sector back out to your *COPYA* disk version of *Pandora's Box*.

Eric Holman Whitaker's APT for...

Castle Wolfenstein

Escape from the Castle Now

In the room that has a stairway at the top of the screen, the following will allow you to escape from the castle immediately:

Open the disk drive door but do not remove the disk. Exit the room via the stairway at the top of the screen. *Castle Wolfenstein* will attempt to save that room on a particular sector, but with the drive door open, this will not be possible. After your disk drive has tried twice to save the room, your man appears above the stairway and his head will be directly to the left. *Do not close the drive door yet, or the game will fry.*

Wait twice more for *Castle Wolfenstein* to try and save the sector and observe what happens: You will have escaped! As soon as you see the escape screen, close the drive door quickly. The picture will load, and if you have the plans, you will receive extra congratulations and a raise in rank.

Robotron

Atarisoft

Deprotecting Robotron
by Clay Harrell
(Hardcore COMPUTIST # 8, page 8)

Requirements:
Apple][, Apple][Plus, Apple //e with 48K
at least one *DOS 3.3* disk drive.
One blank disk
DOS 3.3 System Master disk
Robotron from Atarisoft

Atari is certainly a name that everyone is familiar with when it comes to video games. Atari has successfully marketed other companies' games (after buying the rights, of course) for many home and personal computers. And finally, they have started marketing games for the Apple.

This is good for us, the Apple users, since now we can enjoy many of the favorite arcade games on our Apples. Atari has also blessed us with weak copy-protection, making most of the new Atari releases easily copyable.

Case in point: Robotron.

Robotron 2084 is the best implementation of the William's arcade game I have seen for the Apple. My hat is off to the author of the Apple version, whoever it is (for some reason Atari leaves the author's name out of the game!?). But even though the game is well done, not much time was put into protecting it from prying eyes, especially since copy-protection has evolved so far on the Apple....

Atari uses a slightly modified DOS. This is evident from the conventional boot sounds and the appearance of an Applesoft cursor at the bottom left of the screen when booting the disk.

Just for fun, after booting the disk, type

```
CTRL C
```

This will prevent the BASIC *HELLO* program from running after DOS is loaded on a conventional *DOS 3.3* disk. If you try this on

Robotron, you will find the same thing...the computer beeps, the drive stops spinning and you are placed in Applesoft. You may now list the BASIC program with the command:

LIST

This reveals a one line program that reads:

```
10 HOME: CLEAR: PRINT CHR$(4);"BNROBOTRON"
```

From this program we now know there must be a catalog track, since we can see that some DOS command is used (called BN) to run the file *ROBOTRON*. We know that BN is some kind of DOS command because it is preceeded by CHR$(4).

So naturally, the next thing to do is to type:

CATALOG

Well, we get disappointed with a *SYNTAX ERROR*. The conclusion we can draw from this is that someone at Atari was thinking enough to change the DOS commands from the normal ones (they probably used *DOS Boss* from Beagle Brothers, no doubt).

So the next thing to do is to boot a normal *DOS 3.3* disk and then put the *Robotron* disk in a drive. Now try typing:

CATALOG

This exercise provides us with the rewarding message *I/O ERROR*. This is to be expected. Atari has made the disk uncopyable by changing the epilogue bytes on the disk from DE AA EB to a perverted DE AB FE. This can be seen by using the nibble read commands from either *The Inspector* or *Nibbles Away II* (if you don't have either of these fine utilities, don't worry about it).

Disk format review

For those of you who don't know what 'Epilogue Bytes' are, I will discuss it here for you....

First we must discuss the formatting of a *DOS 3.3* disk. Every normal *DOS 3.3* disk has 35 tracks (0 — 34) and 16 sectors (0 — 15).

How the tracks are located on the disk is hardware dependent, but how the sectors are located is software dependent, hence the name 'soft sectored'. Since software determines the sectoring, it was easy for Apple to change from 13 sector format to 16 sector format back in 1981. With this convenience it is easy to protect the Apple disk format.

For DOS to find the sector it is looking for, it must rely on some road markers. Every sector has what is called an 'Address Field'. The Address Field is a unique set of bytes on every sector that lets DOS know the current disk volume, track, and sector number. It also has a checksum byte to determine if some damage has occurred to the sector making it unreadable.

The unique set of bytes that represent the address field are formatted as such:

prologue	volume	track	sector	checksum	epilogue
D5 AA 96	XX YY	XX YY	XX YY	XX YY	DE AA EB

Whenever DOS sees the unique set of bytes D5 AA 96, it knows the above information follows.

Similarly, there is a 'Data field'. It has its own set of unique bytes to alert DOS to its whereabouts:

prologue	data	checksum	epilog
D5 AA AD	Program, data, etc ...	XX	DE AA EB

Whenever DOS sees the unique set of bytes D5 AA AD, it knows that a program or some kind of data follows. This information is on every sector of a normal *DOS 3.3* disk.

If any one of the prologue bytes are changed, normal DOS would not be able to locate the address or data fields and an *I/O ERROR* would result.

If the epilogue bytes are not what they should be, an *I/O ERROR* will result. This is not due to their uniqueness, since DOS will read whatever two bytes follow the information fields and use them for verification.

Therefore, two easy things to do in protection are to alter the address field and/or the data field prologue bytes, and alter the protected DOS accordingly to locate these unique bytes. Now normal DOS cannot find the address field, so it does not know what sector it is trying to read. Or, it cannot read the data field because it cannot find the unique set of bytes that designates the data. So COPYA will not copy the protected disk.

Altered Epilogue Bytes

What Atari has done is to change the epilogue bytes. This is really a minor change since normal DOS can still find the address field (so it knows what sector it is looking at) and the data field (so it knows where the data is), but you still get an *I/O ERROR* since the epilogue bytes are not what *DOS 3.3* expects to find.

Well, for us to read the disk from normal DOS, all we must do is to defeat the routine that detects errors. If we do this, we will not get an *I/O ERROR* and we will be able to read the disk from normal *DOS 3.3*. The routine that does this lives at $B942. If an error exists, the carry byte is set and DOS says "bad boy" and scolds you with an *I/O ERROR*. So a change at location $B942 from $38 (SEC) to $18 (CLC) will clear the *I/O ERROR* problem!

Now we may CATALOG the *Robotron* disk. Upon doing this, we find two files: *RUNNER* and *ROBOTRON*.

RUNNER is the 'HELLO' program and is unneeded (and also unusable without modification since Atari has changed the DOS commands as follows:

CATALOG has been removed.
BLOAD is **BD**.
BRUN is **BN**.)

So get out *FID* from your *DOS 3.3 System Master* and

```
BRUN FID
```

Now transfer the file *ROBOTRON* to a normal *DOS 3.3* disk and you're all done!

Summary

To re-cap the instructions used to deprotect Robotron:

1 Boot normal *DOS 3.3*.

2 Initialize a disk with normal *DOS 3.3* by typing:

```
FP
INIT HELLO
```

3 To enter the Monitor, type:

```
CALL-151
```

4 Change the error detection routine by typing:

```
B942:18
```

5 Insert your *DOS 3.3 System Master* into a drive and type:

```
BRUN FID
```

6 Use *FID* to transfer the file *ROBOTRON* from the original *Robotron* disk to your freshly INITialized *DOS 3.3* disk.

7 BRUN the file *ROBOTRON* on your normal *DOS 3.3* disk to play *Robotron*.

Sensible Speller version 4.0d

Sensible Software

Sensible Speller Softkey
by Cris Rys
(Hardcore COMPUTIST # 9, page 9)

Requirements:
Apple][, Apple][Plus
16K RAM card
Blank Disk *Sensible Speller V 4.0d*
Super IOB or any DOS track-copy program

Sensible Speller version 4.0 is a very useful dictionary program which has a vocabulary of over 80,000 words. The softkey of this program incorporates a very useful feature of the language card that lets you customize monitor routines so when you press the RESET key, you arrive in the monitor.

Note: If you don't have a language card (separate from the motherboard) you won't be able to use the following procedure.

This softkey contains the following listings:

SPELLER.CON	Applesoft
SPELLER.LOADER	hexdump and source code
SPELLER.SAVER	hexdump and source code

1 With the computer OFF, put your 16K card in slot 1.

2 Boot up the *DOS 3.3 System Master* disk.

3 Enter the Monitor:

```
CALL-151
```

4 Modify DOS so that *SPELLER.LOADER* will work:

```
B639: AD 81 C0 AD 81 C0 4C B3 08
B6B3:A0 00 B9 00 D0
B6B8:99 00 D0 C8 D0 F7 EE B7
B6C0:08 EE BA 08 AD BA 08 D0
B6C8:EC AD 80 C0 A9 07 8D 00
B6D0:02 4C 00 B7
```

5 Initialize the disk you want *Sensible Speller* on with this modified DOS:

```
INIT HELLO
```

6 Insert a different disk and key in the *SPELLER.LOADER* hexdump.

7 Save this program to ease the event of error:

```
BSAVE SPELLER.LOADER,A$B700,L$9A
```

8 Type in this short machine language routine which calls the RWTS:

```
803:A9 B7 A0 E8 4C B5 B7
```

9 Tell the RWTS that we wish to write Track 0, Sector 1 from Page $B7:

```
B7EB:00 00 01
B7F0:00 B7 00 00 02
```

10 Insert the disk you initialized in Step 5 and write the sector:

```
803G
```

11 Now copy Tracks $2—$3 and $6—$22 of your original *Sensible Speller* disk to the disk you initialized in Step 5. These tracks are not protected but you must use a copy program which will copy specific tracks (any bit copier should do). The Super IOB controller (SPELLER.CON) listed at the end of this article will also work.

Here is the routine I mentioned at the beginning of this softkey. *Sensible Speller*, like many other programs, checks for a RAM card in slot 0. Because several programs (such as *Sensible Speller*) only check slot 0, this technique may be used with other programs as well.

12 Boot a normal DOS disk and enter the Monitor:

```
CALL-151
```

13 Write-enable the language card:

```
C091 C091
```

14 Copy the Monitor into the language card:

```
F800<F800.FFFFM
```

15 Read- and write-enable the language card:

```
C093 C093
```

16 Type in the *SPELLER.SAVER* program and save it on the same disk you saved *SPELLER.LOADER* on:

```
BSAVE SPELLER.SAVER,A$D000,L$60
```

17 Type a routine to save Page $Ø into Page $71 when RESET is hit:

```
FA62:A2 ØØ B5 ØØ 9D ØØ 71
E8 DØ F8 4C 59 FF
```

18 Read-enable and write-protect the language card RAM:

```
CØ9Ø
```

19 Insert your original *Sensible Speller* disk and boot it:

```
C6ØØG
```

20 When the menu appears and the disk drive stops running, press:

RESET

You should now be in Monitor.

21 Read-enable the language card in Slot 1:

```
CØ9Ø
```

22 Insert the disk you want *Sensible Speller* on and invoke the *SPELLER.SAVER* program

```
DØØØG
```

The disk drive will start whirling and after a while the cursor will come back.

You're done!

Now may be a good time to put the language card back in Slot Ø, but it is not necessary to do this to run this program.

Don't forget to turn the computer off first.

Super IOB controller...

SPELLER.CON

```
1ØØØ REM SENSIBLE SPELLER HELPER
1Ø1Ø TK = 2:ST = Ø:LT = 35:CD = WR
1Ø2Ø T1 = TK: GOSUB 49Ø
1Ø3Ø GOSUB 43Ø: GOSUB 1ØØ:ST = ST + 1: IF ST < DOS THEN 1Ø3Ø
1Ø4Ø IF BF THEN 1Ø6Ø
1Ø5Ø ST = Ø:TK = TK + 1 + 2 * (TK = 3): IF TK < LT THEN 1Ø3Ø
1Ø6Ø GOSUB 49Ø:TK = T1:ST = Ø
1Ø7Ø GOSUB 43Ø: GOSUB 1ØØ:ST = ST + 1: IF ST < DOS THEN 1Ø7Ø
1Ø8Ø ST = Ø:TK = TK + 1 + 2 * (TK = 3): IF BF = Ø AND TK < LT THEN 1Ø7Ø
1Ø9Ø IF TK < LT THEN 1Ø2Ø
11ØØ HOME : PRINT : PRINT "DONE^WITH^COPY": END
```

hexdump for...

SPELLER.LOADER

```
B7ØØ:   2C 5Ø CØ 2C 57 CØ 2C 52
B7Ø8:   CØ 2C 55 CØ A9 ØF 8D ED
B71Ø:   B7 A9 Ø7 8D EC B7 A2 Ø1
B718:   8E EA B7 CA 8E FØ B7 A9
B72Ø:   5F 8D F1 B7 2Ø 64 B7 CE
B728:   F1 B7 AD F1 B7 C9 4Ø BØ
B73Ø:   F3 A9 3A 8D F1 B7 2Ø 64
B738:   B7 CE F1 B7 AD F1 B7 C9
B74Ø:   Ø8 BØ F3 A9 Ø3 8D F1 B7
B748:   2Ø 64 B7 A9 77 8D F1 B7
B75Ø:   2Ø 64 B7 A9 71 8D F1 B7
B758:   2Ø 64 B7 AD 51 CØ AD 54
B76Ø:   CØ 4C 8A B7 A9 Ø1 8D F4
B768:   B7 A9 B7 AØ E8 2Ø B5 B7
B77Ø:   CE ED B7 1Ø 14 A9 ØF 8D
B778:   ED B7 CE EC B7 AD EC B7
B78Ø:   C9 Ø3 DØ Ø5 A9 Ø1 8D EC
B788:   B7 6Ø A2 ØØ BD ØØ 71 95
B79Ø:   ØØ E8 DØ F8 2Ø 75 32 4C
B798:   D9 33
```

hexdump for...

SPELLER.SAVER

```
DØØØ:   A9 ØF 8D ED B7 A9 Ø5 8D
DØØ8:   EC B7 A2 Ø1 8E EA B7 CA
DØ1Ø:   8E FØ B7 A9 3A 8D F1 B7
DØ18:   2Ø 3A DØ CE F1 B7 AD F1
DØ2Ø:   B7 C9 Ø8 BØ F3 A9 Ø3 8D
DØ28:   F1 B7 2Ø 3A DØ A9 77 8D
DØ3Ø:   F1 B7 2Ø 3A DØ A9 71 8D
DØ38:   F1 B7 A9 Ø2 8D F4 B7 A9
DØ4Ø:   B7 AØ E8 2Ø B5 B7 CE ED
DØ48:   B7 1Ø 14 A9 ØF 8D ED B7
DØ5Ø:   CE EC B7 AD EC B7 C9 Ø3
DØ58:   DØ Ø5 A9 Ø1 8D EC B7 6Ø
```

SPELLER.LOADER

```
1000                                    *_________________________
1010                                    * SPELLER.LOADER
1020                                    * THIS LOADS WHAT USED TO BE
1030                                    * PART OF SENSIBLE SPELLER
1040                                    * WHILE IT IS BOOTING
1050                                    *_________________________
1060
1070                 .OR $B700
1080                 .TFSPELLER.LOADER
1090
1100 DRIVE           .EQ $B7EA
1110 TRACK           .EQ $B7EC
1120 SECTOR          .EQ $B7ED
1130 BUFHI           .EQ $B7F1
1140 COMMAND         .EQ $B7F4
1150 RWTS            .EQ $B7B5
1160                 BIT $C050
1170                 BIT $C057
1180                 BIT $C052
1190                 BIT $C055
1200                 LDA #$F
1210                 STA SECTOR
1220                 LDA #$7
1230                 STA TRACK
1240                 LDX #1
1250                 STX DRIVE
1260                 DEX
1270                 STX BUFHI-1
1280                 LDA #$5F
1290                 STA BUFHI
1300 LOOP1           JSR READ
1310                 DEC BUFHI
1320                 LDA BUFHI
1330                 CMP #$40
1340                 BCS LOOP1
1350                 LDA #$3A
1360                 STA BUFHI
1370 LOOP2           JSR READ
1380                 DEC BUFHI
1390                 LDA BUFHI
1400                 CMP #8
1410                 BCS LOOP2
1420                 LDA #$3
1430                 STA BUFHI
1440                 JSR READ
```

```
1450                LDA #$77
1460                STA BUFHI
1470                JSR READ
1480                LDA #$71
1490                STA BUFHI
1500                JSR READ
1510                LDA $C051
1520                LDA $C054
1530                JMP EXIT
1540 READ           LDA #1
1550                STA COMMAND
1560                LDA #$B7
1570                LDY #$E8
1580                JSR RWTS
1590                DEC SECTOR
1600                BPL RTS1
1610                LDA #$F
1620                STA SECTOR
1630                DEC TRACK
1640                LDA TRACK
1650                CMP #$3
1660                BNE RTS1
1670                LDA #1
1680                STA TRACK
1690 RTS1           RTS
1700 EXIT           LDX #0
1710 LOOP3          LDA $7100,X
1720                STA $0,X
1730                INX
1740                BNE LOOP3
1750                JSR $3275
1760                JMP $33D9
```

source code for...

SPELLER.SAVER

```
1000                                 *______________________________
1010                                 * SPELLER.SAVER
1020                                 * THIS SAVES THE SENSIBLE
1030                                 * SPELLER FROM MEMORY TO
1040                                 * VARIOUS SECTORS OF AN
1050                                 * UNPROTECTED DISK
1060                                 *______________________________
1070
1080                .OR $D000
1090                .TF SPELLER.SAVER
```

```
11ØØ
111Ø DRIVE       .EQ $B7EA
112Ø TRACK       .EQ $B7EC
113Ø SECTOR      .EQ $B7ED
114Ø BUFHI       .EQ $B7F1
115Ø COMMAND     .EQ $B7F4
116Ø             LDA #$F
117Ø             STA SECTOR
118Ø             LDA #$5
119Ø             STA TRACK
12ØØ             LDX #1
121Ø             STX DRIVE
122Ø             DEX
123Ø             STX BUFHI-1
124Ø             LDA #$3A
125Ø             STA bUFHI
126Ø LOOP        JSR WRITE
127Ø             DEC BUFHI
128Ø             LDA BUFHI
129Ø             CMP #$8
13ØØ             BCS LOOP
131Ø             LDA #$3
132Ø             STA BUFHI
133Ø             JSR WRITE
134Ø             LDA #$77
135Ø             STA BUFHI
136Ø             JSR WRITE
137Ø             LDA #$71
138Ø             STA BUFHI
139Ø WRITE       LDA #2
14ØØ             STA COMMAND
141Ø             LDA #$B7
142Ø             LDY #$E8
143Ø             JSR $B7B5
144Ø             DEC SECTOR
145Ø             BPL RTS1
146Ø             LDA #$F
147Ø             STA SECTOR
148Ø             DEC TRACK
149Ø             LDA TRACK
15ØØ             CMP #$3
151Ø             BNE RTS1
152Ø             LDA #1
153Ø             STA TRACK
154Ø RTS1        RTS
```

Sensible Speller 4*

**Sensible Speller Version 4.0c & 4.1c*

Sensible Software

More On Sensible Speller 4
by Lamont Cranston
(Hardcore COMPUTIST # 10, page 6)

Requirements:
Apple][, Apple][Plus
Blank disk
Bit Copy program
Sector Editor

Editors Note: The previous *Sensible Speller* Softkey was based upon revision 4.0d of the program. We verified the softkey using that revision. Several readers however reported that the softkey would not work on their version of the program. Apparently there are at least **six** different revisions, each having a different menu entry point. We are trying to gather information on the different versions, but as yet we do not have a verified procedure that will work on anything other than 4.0d.

Because of the interest in *Sensible Speller* we are publishing a softkey for revisions 4.0c and 4.1d. This method has not been verified by our staff.

The protection scheme on *Sensible Speller 4* is based upon the disks having a very strange and complicated boot. The complexity of the boot requires a lot of extra code which apparently exists solely to handle the disk's odd format.

Tracks $0—$1 and $4—$5 are used.

Tracks $2—$3 and $6—$22 are all normal *DOS 3.3* format (Track $3 is technically *CP/M*, but *COPYA* will still copy it).

The title picture resides on Tracks $6—$7 and is loaded on hi-res page 2 at $4000—$5FFF.

Track $8—$B contain the *SS* main menu routines.

Other routines, such as the spelling checker itself, are loaded later by the menu module.

Knowing this, all we have to do is load the menu program with its associated support routines and fix its expected environment.

Since the useful program exists on normal *DOS 3.3* tracks, we will write Track $Ø (on a copy of the *SS* disk) with the *DOS 3.3* BOOT1 and RWTS sectors, then rewrite the DOS BOOT2 (Track $Ø, Sector $1) routine that's loaded at $B7ØØ as part of RWTS so that it loads in the *Sensible Speller* menu.

A couple of routines also have to be added to BOOT2 so that the main menu will work properly once loaded.

BOOT2 is called from BOOT1 after the RWTS has been loaded. The *SS* menu program loads and runs at $8ØØ—$47FF. The BOOT2 code needed for *SS* has to do the following things:

1 Load the *SS* menu program in memory at $8ØØ—$47FF.

2 Load Track $2, Sectors $E—$F in memory at $8ØØ—$9FF.

3 Store a $ØØ in the prompt register at $33.

4 Copy the Motherboard *F8 Monitor ROM* into a language card if it is present.

5 Set normal I/O.

6 Exit to the entry point of the *SS* menu.

Step 3 is necessary because *Sensible Speller* does not seem to work correctly if a prompt is found at $33.

Step 4 is needed becuase *SS* will crash if a language card without a monitor image is found in slot Ø.

The DOS BOOT2 loader is well documented in the book: **Beneath Apple DOS**. Essentially it is designed to load a specified number of sectors into memory from consecutive disk sectors into consecutive pages of memory. The Input/Output Block (IOB) and Device Control Table (DCT) are present at the end of the BOOT2 sector.

What version?

I have used this patch on *Sensible Speller* version 4.Øc and 4.1c. The source code listing is for version 4.1c (the *PFSWrite*-compatible version). The 4.Øc version (the Pascal-compatible version) menu has a different entry point ($33B8), so the JMP in BOOT2 at $B79Ø (lines 1070 and 1790 in source code) must be changed accordingly.

If you have an earlier version of *Sensible Speller* you can use the *Inspector* to search for the entry. To do this, boot up the *Inspector*, set the buffer to $8ØØ and read Track $8, Sector $Ø and then continue holding down the I key until you have filled memory from $8ØØ—$47FF with consecutive sectors from the disk. Search the buffer for a likely entry point and use it to put into the modified BOOT2 code. A conspicuous feature of the entry is that it will store its address into $3E—$3F.

The modified BOOT2 code will not load the hi-res *Sensible Speller* logo because this would make the code longer than one sector, although it could be rewritten to load in a second sector.

Sectors $E—$F of Track $2 are loaded over the menu code because this is where *SS* stores its configuration data. These sectors can be read by any sector editor.

The VTOC on the copied disk can be altered to free up Track $1 and Track $4—$5 for data since they are not used by *Sensible Speller*. If you wish to implement an extended boot to load DOS or the hi-res picture, the extra data can be stored on these sectors or on the unused sectors of Track $Ø.

The Necessary Steps

1 Boot up *DOS 3.3* and INIT a slave disk:

```
PR#6
FP
INIT HELLO
```

2 Use a bit copier to copy Tracks $2—$3 and $6—$22 from the *Sensible Speller* original onto the disk initialized in Step 1.

3 Use a sector editor to write the modified BOOT2 code onto Track $Ø, Sector $1 of the copied disk. Do not forget to change the entry point if you have a version other than 4.1c.

4 That's all. Now start misspelling!

I was motivated to deprotect *Sensible Speller* because of Sensible Software's heavy protection of this utility program (probably soon to get heavier), its serial numbering and restrictive licensing agreement.

Other factors which motivated me were *Sensible Speller*'s refusal to boot on an Apple //c (the computer rightly says there is something wrong with the disk) and *Sensible Speller*'s withdrawal of *Pascal* support with the release of version 4.1c. To me, it seems as if dropping an entire operating system in ́favor of *Word Handler* or *Cameo* seems to be a ploy to get those coveted serial number registration cards. All in all, they should have spent the money they spent on copy-protection on some extra features for the program.

I did not find the program particularly difficult to crack (about 6 hours work all told). I want to emphasize that I **did buy** the original program and do not condone piracy. However, nobody locks me out of a program I have purchased, either.

BOOT2

```
B7ØØ:  8E E9 B7 8E F7 B7 A9 Ø1
B7Ø8:  8D F8 B7 8D EA B7 AD EØ
B71Ø:  B7 8D E1 B7 A9 ØB 8D EC
B718:  B7 A9 ØF 8D ED B7 AØ 48
B72Ø:  88 8C F1 B7 A9 Ø1 8D F4
B728:  B7 8A 4A 4A 4A 4A AA A9
B73Ø:  ØØ 85 33 9D F8 Ø4 9D 78
B738:  Ø4 2Ø 93 B7 A9 Ø1 8D F8
B74Ø:  B7 8D EA B7 A9 Ø2 8D E1
B748:  B7 A9 Ø2 8D EC B7 A9 ØF


B75Ø:  8D ED B7 AØ ØA EA 88 8C
B758:  F1 B7 A9 Ø1 8D F4 B7 8C
B76Ø:  F1 B7 A9 Ø1 8D F4 B7 2Ø
B768:  93 B7 A2 FF EA 8E EB B7
B77Ø:  2Ø 93 FE 2Ø 89 FE AD 81
B778:  CØ AD 81 CØ A9 F8 AØ FF
B78Ø:  84 3E 84 3F C8 84 42 84
B788:  3C 85 43 85 3D 2Ø 2C FE
B79Ø:  4C 17 35 AD E5 B7 AC E4
B798:  B7 2Ø B5 B7 AC ED B7 88


B7AØ:  1Ø Ø7 AØ ØF EA EA CE EC
B7A8:  B7 8C ED B7 CE F1 B7 CE
B7BØ:  E1 B7 DØ DF 6Ø Ø8 78 2Ø
B7B8:  ØØ BD BØ Ø3 28 18 6Ø 28
B7CØ:  38 6Ø AD BC B5 8D F1 B7
B7C8:  A9 ØØ 8D FØ B7 AD F9 B5
B7DØ:  49 FF 8D EB B7 6Ø A9 ØØ
B7D8:  A8 91 42 C8 DØ FB 6Ø ØØ
B7EØ:  4Ø ØØ ØA 1B E8 B7 ØØ B6
B7E8:  Ø1 6Ø Ø1 ØØ ØØ ØØ FB B7


B7FØ:  ØØ 47 ØØ FF Ø1 ØØ FE 6Ø
B7F8:  Ø1 ØØ ØØ ØØ Ø1 EF D8 ØØ
```

BOOT2 Source code

```
1ØØØ MEM1         .EQ $Ø4F8  Last Track read by DOS
1Ø1Ø MEM2         .EQ $Ø478  Next Track to be read
1Ø2Ø READ.PGS     .EQ $B793
1Ø3Ø SETVID       .EQ $FE93
1Ø4Ø SETKBD       .EQ $FE89
1Ø5Ø LANG.CARDØ   .EQ $CØ8Ø  START OF MAGIC MEMORY LOCATIONS THAT
                             AFFECT EXTRA 16K OF MEMORY IN SLOT Ø
1Ø6Ø MOVE         .EQ $FE2C  MONITOR "M" SUBROUTINE
1Ø7Ø START.SPELL  .EQ $3517
1Ø8Ø RWTS         .EQ $BDØØ  ENTRY TO RWTS
1Ø9Ø FM.PRM       .EQ $B5BB  FILE MANAGER PARAMETER LIST
11ØØ FM.VOL       .EQ $B5F9  FILE MANAGER VOL NUMBER COMPLIMENTED
111Ø STACK        .EQ $1ØØ   STACK AREA
112Ø
113Ø              .OR $B7ØØ
114Ø              .TF SSBOOT2
115Ø
116Ø              STX RWTS.PRM+1    NEXT SLOT
117Ø              STX RWTS.PRM+15   LAST SLOT
118Ø              LDA #1
119Ø              STA RWTS.PRM+16   NEXT DRIVE
12ØØ              STA RWTS.PRM+2    LAST DRIVE
121Ø              LDA SECND.PRM     NUMBER OF PAGES
122Ø              STA SECND.PRM+1   NUMBER OF SECTORS
123Ø              LDA #$ØB
124Ø              STA RWTS.PRM+4    TRACK NUMBER
125Ø              LDA #$ØF
126Ø              STA RWTS.PRM+5    SECTOR NUMBER
127Ø              LDY #$48
128Ø              DEY
129Ø              STY RWTS.PRM+9    BUFFER MSB
13ØØ              LDA #1
131Ø              STA RWTS.PRM+12   COMMAND CODE
132Ø              TXA               A<=SLOT NUMBER FOUND*16
133Ø              LSR               DIVIDE SLOT
134Ø              LSR               NUMBER BY
135Ø              LSR               16
136Ø              LSR
137Ø              TAX               CORRECT OFFSET
138Ø              LDA #Ø
139Ø              STA $33
14ØØ              STA MEM1,X        LAST TRACK READ WAS PHASE Ø
141Ø              STA MEM2,X        DON'T MOVE HEAD ANY
142Ø              JSR READ.PGS
143Ø              LDA #1
```

```
144Ø                STA RWTS.PRM+16   LAST DRIVE
145Ø                STA RWTS.PRM+2    LAST DRIVE
146Ø                LDA #2
147Ø                STA SECND.PRM+1   NUMBER OF SECTORS
148Ø                LDA #2
149Ø                STA RWTS.PRM+4    TRACK NUMBER
15ØØ                LDA #$ØF
151Ø                STA RWTS.PRM+5    SECTOR NUMBER
152Ø                LDY #$ØA
153Ø                NOP
154Ø                DEY
155Ø                STY RWTS.PRM+9    BUFFER MSB
156Ø                LDA #1
157Ø                STA RWTS.PRM+12   COMMAND CODE
158Ø                STY RWTS.PRM+9    BUFFER MSB
159Ø                LDA #1
16ØØ                STA RWTS.PRM+12   COMMAND CODE
161Ø                JSR READ.PGS
162Ø                LDX #$FF
163Ø                NOP
164Ø                STX RWTS.PRM+3    VOLUME EXPECTED = 255
165Ø                JSR SETVID        PR#Ø
166Ø                JSR SETKBD        IN#Ø
167Ø                LDA LANG.CARDØ+1
168Ø                LDA LANG.CARDØ+1
169Ø                LDA #$F8
17ØØ                LDY #$FF
171Ø                STY $3E
172Ø                STY $3F
173Ø                INY
174Ø                STY $42
175Ø                STY $3C
176Ø                STA $43
177Ø                STA $3D
178Ø                JSR MOVE          F8ØØ<F8ØØ.FFFFM
179Ø                JMP START.SPELL
18ØØ NXT.SECT       LDA SECND.PRM+5   RWTS POINTER MSB
181Ø                LDY SECND.PRM+4   RWTS POINTER LSB
182Ø                JSR RWTS1
183Ø                LDY RWTS.PRM+5    SECTOR NUMBER
184Ø                DEY               DONE WITH TRACK?
185Ø                BPL NXT.TRK       NO,BRANCH
186Ø                LDY #$ØF          NEXT TRACK, SECTOR 15
187Ø                NOP
188Ø                NOP
189Ø                DEC RWTS.PRM+4    TRACK NUMBER
19ØØ NXT.TRK        STY RWTS.PRM+5    SECTOR NUMBER
191Ø                DEC RWTS.PRM+9    BUFFER MSB
192Ø                DEC SECND.PRM+1   NUMBER OF SECTORS
193Ø                BNE NXT.SECT
```

```
194Ø            RTS
195Ø RWTS1      PHP
196Ø            SEI
197Ø            JSR RWTS
198Ø            BCS RWTS.ERR
199Ø            PLP
2ØØØ            CLC
2Ø1Ø            RTS
2Ø2Ø RWTS.ERR   PLP
2Ø3Ø            SEC
2Ø4Ø            RTS
2Ø5Ø SET.WRIT   LDA FM.PRM+1      SET RWTS
2Ø6Ø            STA RWTS.PRM+9    FOR WRITING
2Ø7Ø            LDA #Ø
2Ø8Ø            STA RWTS.PRM+8    RWTS BUFFER LSB
2Ø9Ø            LDA FM.VOL
21ØØ            EOR #$FF
211Ø            STA RWTS.PRM+3    VOLUME EXPECTED
212Ø            RTS
213Ø ZERO.BUF   LDA #Ø            ZERO CURRENT BUFFER
214Ø            TAY
215Ø .1         STA ($42),Y       LOCATION $42 HOLDS THE
                                  POINTER TO CURRENT BUFFER
216Ø            INY
217Ø            BNE .1
218Ø            RTS
219Ø            .HS ØØ            UNUSED
22ØØ SECND.PRM  .HS 4ØØØØA1BE8B7ØØB6
                                  DOS SECOND STAGE BOOT LOADER
                                  PARAMETER LIST
221Ø RWTS.PRM   .HS Ø16ØØ1ØØØØØØFBB7ØØ47ØØFFØ1ØØFE6ØØ1
222Ø * RWTS.PRM IS THE RWTS PARAMETER LIST
223Ø            .HS ØØØØ          UNUSED
224Ø DCT        .HS ØØØ1EFD8      DEVICE CHARACTERISTICS
                                  TABLE
225Ø            .HS ØØ            UNUSED
```

The Spy Strikes Back

Penguin Software

Backing-Up The Spy Strikes Back
by Clay Harrell
(Hardcore COMPUTIST # 8, page 6)

Requirements:
48K Apple with old style F8 Monitor ROM
One disk drive with *DOS 3.3*
Initialized 48K *DOS 3.3* slave disk
The Spy Strikes Back

Penguin Software has started what I consider to be the best idea since canned beer: inexpensive software!

It is at the point where backing up software is not really necessary since the cost of Penguin's games are under 2Ø dollars!

But there is a catch to *The Spy Strikes Back*: Penguin has a contest in which the first person in each state who can solve their puzzle will win $1ØØ in software. This means code-snooping is a must (I'm lazy I guess) and deprotection is along this line.

Upon booting *The Spy* disk, I notice that the boot is very short. It seems to load only six or seven tracks before the game starts. The game has some disk access within it for high scores and the like, and to discourage the *Wildcard/Replay Card* crowd from getting easy copies.

After loading the game and hitting RESET, I notice that both hi-res pages are being used for the graphics (therefore, no code exists there).

Keeping in mind the short load, snoop through memory looking for a starting location. The typical starting location for a game that uses both hi-res pages is $8ØØ or $6ØØØ.

Sure enough, a **6ØØØG** from the Monitor starts the game up with the disk spinning to load the hi-scores.

Further examination of memory reveals that $18ØØ—$1FFF is the code to draw the pictures for the demo and that the two hi-res pages do not have to be saved. Locations $6ØØØ—$9AØØ seem to be the actual code for the game.

Note: a good way to see what part of memory is being used is to zero-out all of memory and then boot the program and see what is used.

For example, from Monitor type:

```
800:0 N 801<800.BFFFM
```

This will wipe out memory from $800 to $BFFF with zeroes. This was the method used to determine what areas of memory *The Spy Strikes Back* used. (I assumed that RWTS lived at it's normal location of $B700).

Now that we know where *The Spy* lives, we must clear the way for a slave disk boot.

Recall that a slave disk does not destroy memory at $900—$95FF. Also recall that we do not have to save the two hi-res pages (locations $2000—$5FFF) since the game re-draws these for us using the data at $1800—$1FFF.

This means we can move the code at $9600—$9A00 to $1000 so not to destroy it when we boot our slave disk.

Another thing to consider is the disk access in the game. I have done the work for you and have found the locations to change to remove this pesty protection.

With these things in mind, the exact procedure for backing-up *The Spy Strikes Back* is as follows:

1 Boot the original disk *The Spy Strikes Back*:

```
PR#6
```

2 RESET into the monitor when the demo starts.

3 Move the code at $9600—$9A00 to $1000 where it is safe during a slave boot:

```
1000<9600.9A00M
```

4 Bypass the disk access routines:

```
101F:EA EA EA
```

5 Boot a 48K DOS 3.3 slave disk:

```
PR#6
```

6 Set MAXFILES to 1:

```
MAXFILES1
```

7 Enter the Monitor:

```
CALL -151
```

8 Move the code saved in step 3 back to $9600:

```
9600<1000.1400M
```

9 BSAVE the first portion of code:

```
BSAVE SPY2,A$6000,L$3A00
```

10 BSAVE the second portion of code:

```
BSAVE SPY1,A$1800,L$800
```

11 Re-enter BASIC:

```
3D0G
```

12 Clear any Applesoft program in memory:

```
FP
```

13 Create the following Applesoft program:

```
10 HIMEM:24576
20 D$ = CHR$(4)
30 PRINT D$;“MAXFILES1”
40 PRINT D$;“BLOAD SPY2”
50 PRINT D$;“BLOAD SPY1”
60 FOR I = 0 to 14: POKE 768 + I,32: NEXT:
     FOR I = 15 to 254: POKE 768 + I,0: NEXT
70 CALL 24576
```

14 SAVE the Applesoft program:

```
SAVE SPY STRIKES BACK
```

Final notes: The code that you changed at location $101F takes out the disk access in the game. The POKE statements at location 768 set up the high scores as zero with no one's initials.

Good luck in finding the hints to the puzzle (you didn't expect me to tell you how to find them, did you??).

Visible Computer: 65Ø2

Software Masters

Softkey for The Visible Computer: 65Ø2
by Jared Block and Bob Bragner
(Hardcore COMPUTIST # 9, page 26)

Requirements:
Super IOB program
Blank disk
The Visible Computer program

The Visible Computer 65Ø2 is an excellent educational program designed to teach 65Ø2 machine language to those would-be Bill Budges among us. The program allows the user to step and trace through machine language programs, displaying the effects upon registers and memory in hi-res graphics. The package comes with very good documentation but unfortunately, the disk is copy-protected. With the following procedure *The Visible Computer 65Ø2* can be copied and modified to work with normal *DOS 3.3*.

The Visible Computer uses a modified DOS as its primary form of protection. In addition to the DOS marks being altered and the encoding manner, most of the DOS commands have been obliterated. Because of a few mistakes by the creators of *The Visible Computer*, we can easily capture its strange RWTS and use it to copy the disk. Once a copy has been made, the main program needs a few modifications in order to work with *DOS 3.3*.

Getting Turned On

Start by turning *ON* your computer. Next, place your original version of *Visible Computer* in the disk drive but DO NOT CLOSE THE DRIVE DOOR.

Press:

CTRL C

and then shut the door. In a few seconds you will get a *BREAK* message and a prompt.

Right away I thought I was 'in' *The Visible Computer*. But the RUN flag gets set during *The Visible Computer*'s boot process. Therefore, anything you type will execute the program in memory (never to give control back to you).

This is where we take advantage of their first mistake. We type:

```
FP
```

which clears the program in memory but more importantly, clears the RUN flag.

Making The Invisible Visible

Now you are 'in' *The Visible Computer*. Sure the DOS commands have been obliterated, but you can still call them directly. Try a

```
CALL 42350
```

to see the CATALOG.

What we have to do now is save the RWTS portion of the DOS in memory so that Super IOB can use it. This is accomplished by the Monitor command

```
1900<B800.BFFFM
```

Next, boot your Super IOB disk and save the RWTS with

```
BSAVE VISICOMP.RWTS,A$1900,L$800
```

With this file, all you have to do is use the Super IOB controller.

Once the copy has been made, there are a couple of modifications and deletions that must be made to the main Applesoft program called *The VC* so that it will work properly with *DOS 3.3*.

The first deletion is line 1 of the program. This line is a REM statement containing a CTRL M a CTRL D and the DOS command **FP**. This has the effect that if the line 1 is listed, DOS will process the FP command and delete the program. This problem is easily rectified by deleting Line 1.

So Long SOREN

The BSAVE command in *The Visible Computer*'s DOS has been changed to **SOREN**. The only place this command appears is in Line 13910. Of course, the normal *DOS 3.3* command BSAVE must be restored to this line.

Line 18000 determines whether the program can Read and Write to normal *DOS 3.3*, depending upon the setting of *The Visible Computer*'s 'MASTER' command. This line contains two CALLs to *The VC*'s modified DOS which need to be removed. With these two CALLs removed, the 'MASTER' command will still function normally except that the program will not be able to read files from the original *Visible Computer* program disk.

Finally, and most importantly, Line 20730 contains a CALL that will INIT the disk if the program is used under normal DOS. The INIT command has been removed from *The Visible Computer*'s

DOS, so the CALL is harmless until the program is run under *DOS 3.3*. This nasty little line must be completely deleted.

Here's how to deprotect *The Visible Computer*.

Revealing The Visible Computer

1 Turn on the computer and tell it to break:

CTRL C

2 Insert *The Visible Computer* disk and shut the drive door.

3 Clear the RUN flag:

FP

4 Enter the Monitor:

CALL-151

5 Move the RWTS to a safe place:

1900<B800.BFFFM

6 Boot a 48K slave disk with a short *HELLO* program:

C600G

7 Save the RWTS:

BSAVE VISICOMP.RWTS,A$1900,L$800

8 Load Super IOB and type in the controller, then:

SAVE SUPER IOB.VISICOMP

9 Copy *The Visible Computer*:

RUN

10 Load the main program from the copied disk:

LOAD VC

11 Delete Line 1 so the program can be listed normally:

1

12 LIST Line 13910. It should read:

```
13910 VTAB 20: PRINT D$"SOREN"P2$Z$
```

Change this line so it has a normal BSAVE command:

```
13910 VTAB 20: PRINT D$"BSAVE"P2$Z$
```

13 Line 18ØØØ contains some CALLs to *The VC*'s DOS which must be removed. This line now reads:

```
18ØØØ  PV = T: CALL 47741: IF P2$ = "OFF" THEN PV = F: CALL 47741: REM
    SET NRML,PRTCT DOS'S
```

Change this line so it reads:

```
18ØØØ  PV = T: IF P2$ = "OFF" THEN PV = F: REM SET MASTER ON/OFF
```

14 Line 2Ø73Ø contains a CALL which will INIT normal *DOS 3.3* disks. It reads:

```
2Ø73Ø CALL PEEK (4Ø222) + PEEK(4Ø223) * Q4 + 1
```

Be sure to get rid of it by typing:

```
2Ø73Ø
```

15 If you wish you can also have the program set the Reset Vector so that the program does not reRUN itself when the RESET key is pressed. For instance, if you want to enter the Monitor when RESET is pressed change line 2 to read:

```
2 POKE 1Ø1Ø,89: POKE 1Ø11,255: CALL-1169
```

16 Finally SAVE the modified program back to the copied disk

```
SAVE VC
```

As a CATALOG should reveal, this disk is completely broken and can be copied with *COPYA* or even *FID*. Frequent sessions with *The VC* are recommended for aspiring bit-brains. Good luck!

Super IOB Controller...

SPELLER.CON

```
1ØØØ REM SWAP CONTROLLER (VISIBLE COMPUTER)
1Ø1Ø TK = 3:ST = Ø:LT = 35:CD = WR
1Ø2Ø T1 = TK: GOSUB 49Ø: GOSUB 36Ø
1Ø3Ø GOSUB 43Ø: GOSUB 1ØØ:ST = ST + 1: IF ST < DOS THEN 1Ø3Ø
1Ø4Ø IF BF THEN 1Ø6Ø
1Ø5Ø ST = Ø:TK = TK + 1: IF TK < LT THEN 1Ø3Ø
1Ø6Ø GOSUB 49Ø:TK = T1:ST = Ø: GOSUB 36Ø
1Ø7Ø GOSUB 43Ø: GOSUB 1ØØ:ST = ST + 1: IF ST < DOS THEN 1Ø7Ø
1Ø8Ø ST = Ø:TK = TK + 1: IF BF = Ø AND TK < LT THEN 1Ø7Ø
1Ø9Ø IF TK < LT THEN 1Ø2Ø
11ØØ HOME : PRINT "EVERYTHING^O.K.^DOS^NOT^COPIED": END
1ØØ1Ø  IF  PEEK  (64ØØ)<>  162  THEN  PRINT  CHR$  (4)"BLOAD
    VISICOMP.RWTS,A$19ØØ"
```

Visidex

Visicorp

Backing up Visidex

by Anthony Barnett

(Hardcore COMPUTIST # 9, page 7)

Requirements:

48K Apple
One disk drive with *DOS 3.3*
Visidex
One blank disk
COPYA

Visidex is a key-word index program which has some data storage and retrieval procedures which I have not seen in any other program for the Apple. Its retrieval of data by keyword is very fast.

In Hardcore COMPUTIST # 3 page 6 (see the Book Of Softkeys volume I) in my softkey for *Visitrend / Visiplot*, I explained the problems in obtaining backups from Visicorp when overseas. I notice in Hardcore COMPUTIST # 5 that Bob Bragner describes similar problems in his softkey for *Visifile* (also see the Book Of Softkeys Volume 1). The same problems apply to *Visidex*.

Though I have been able to copy *Visidex* using the long list of *Locksmith* parameters, I decided that an unprotected backup would be more convenient. For one thing, the disk could also be used to store other files instead of being wasted on one 17K file.

During the course of my examination of the *Visidex* disk, I discovered that unless the DOS from the original disk is present, *Visidex* data disks may become corrupted. I also noticed that apart from the dummy serial number, there appeared to be only two very short binary files, namely *VISIDEX* and *VISIO.8*. It turns out that *VISIDEX* is a loader file which uses an RWTS routine to load the whole program into memory.

After some experimentation, I found the following procedure to work.

1 Use *COPYA* to make a copy of *Visidex*.

2 **UNLOCK** and **DELETE** the file *VISIDEX* on the copy.

3 Boot the copy and wait for the *FILE NOT FOUND* message.

4 Insert the original and:

```
BLOAD VISIDEX
```

5 With the original still in drive 1, enter the Monitor with:

```
CALL -151
```

6 From the Monitor type:

```
60A3:69 FF
6000G
```

7 When the drive stops, you should still be in the Monitor. Remove the original and insert the copy.

8 Type:

```
INIT VISIDEX
```

9 When initialization is complete, type:

```
DELETE VISIDEX
BSAVE VISIDEX,A$803,L$4404
```

The disk so produced, if booted, will run *Visidex*. Steps 1 to 3 enable you to get *Visidex*'s DOS into memory. Step 4 gets the program loader into memory. Steps 5 and 6 cause a jump to the monitor before the *Visidex* program is run. The technique here is the same as used with boot code tracing. At $60A2 is the jump out from the loader to the *Visidex* program at $1803. Step 6 alters this to a jump to $FF69 which is the Monitor entry address.

During the course of the load, the file *VISIO.8* is loaded at $4C00. The essence of this file is 6 NOP's and one RTS. Thus, the effective program begins at $803 and ends at $4C06.

Step 8 is necessary because a *DISK FULL* message is received if you attempt to BSAVE the program at this point. Be careful not to INIT the original. It is a good idea to keep a write protect tab on original program disks.

Step 9 prevents a *FILE TYPE MISMATCH* error when BSAVEing the program.

The only difference between this version and the original that I can detect is the response to a RESET. In the original, this usually took you to a *BOOT FROM SLOT 6* message. In the copy, RESET dumps you back to Applesoft. For those of you who want to make patches to the program, be warned. The program makes extensive use of 'funny jumps', i.e. pushing an address onto the stack and then doing an RTS to get there. That's it!

Visiterm

Visicorp

Short Softkey For Visiterm
by B. Baker
(Hardcore COMPUTIST # 9, page 8)

Requirements:
Apple][family with 48K
One blank disk
COPYA
A sector editing program

1 Copy the disk, using *COPYA*.

2 Use the sector editor to read Track $15, Sector $0E of the backup disk.

3 Change Byte $DF from $B0 to $90 and write the sector back to the disk.

Donald Oliveau's APT for...

Wizard and the Princess

Return To Life

If the player 'dies' in this game, there's a way to revive him or her. When the computer asks you *DO YOU WISH TO PLAY AGAIN?* after the 'death' has occurred, answer 'N' for NO and then press RETURN TWICE. This restores you to life at the spot you were 'killed' with all your inventory intact.

Zaxxon

Datasoft,Inc.

Deprotecting Zaxxon
by Clay Harrell
(Hardcore COMPUTIST # 7, page 8)

Requirements:
48K Apple or Apple][Plus
One disk drive and *DOS 3.3*
COPYA from *3.3 System Master*
A sector editor such as *Zap*
Two blank disks
Original *Zaxxon* disk

One of my favorite arcade games of all time is *Zaxxon*. The game was the first to use realistic appearing three-dimensional graphic effects that played well. I've spent a great deal of time (and tokens) playing the game, so naturally, when it was available for the Apple, I bought it.

The only complaint I have about *Zaxxon* is that there are only three planes given for each game. The arcade *Zaxxon* I had always played had five. I felt cheated! With this in mind, I decided to dig into the code that made *Zaxxon* fly.

(Please note: There has been a new release of *Zaxxon* from Datasoft. This is the one I will be primarily discussing, although the same protection was used on both versions. You can tell if you have the new version because it has an option to use the *Mockingboard* with it. If your *Zaxxon* boots and asks *MOCKINGBOARD IN SLOT 4 Y/N?*, you have the new version. In addition, I have seen two different protections used on the newer versions of *Zaxxon*. I will give details on the deprotection of all versions, including the older version of the game).

The first thing to notice is the boot of *Zaxxon*. Listen to your disk drive as the game boots and you can hear the drive arm swing out to an outside track and then swing back in and read the game in. This is what is loosely known as a 'nibble count' or a 'checksum' routine.

If a byte doesn't match the benchmark like it should when the outside track is read, the game will clear memory and reboot. Usually, the only thing involved in deprotecting a program that isn't a single load and that has a 'nibble count' is to find the routine that does the check and jump around it. Usually, too, about the only way to find this routine is to (yech!) trace the boot.

Boot Code Tracing

Boot code tracing is a method of tracing how a program gets from your disk to memory. It does not magically happen all at once but in stages which we can trace and examine and, hopefully, understand. Hence, the name 'boot code tracing.'

The theory of boot code tracing suggests that you should follow the boot process one step at a time to see where it takes you, by altering the code to prevent it from running away from you. Yes, it is advisable that you understand Assembly language, since the code that boots the disk is, in many cases, intentionally misleading.

This process is based upon the law that Track $Ø, Sector $Ø, must always be read by the disk controller card into page 8 ($8ØØ—$8FF) of memory. After this, depending on the complexity of the protection, it is sometimes difficult to understand what goes on in the rest of the load.

In a normal slave disk boot, there are three stages of a boot, starting with the code at $C6ØØ—$C6FF in the disk controller card. This controller card code loads in Track $Ø, Sector $Ø, into $8ØØ—$8FF, which, in turn, loads in Track $Ø, Sectors $Ø—$9, into $B6ØØ—$BFFF. This new code loads in Track $Ø, Sector $C through Track $2, Sector $4, into $9DØØ—$B5FF and finally your *Hello* program is run.

Boot Stages

Boot Stage	Code Location	Final Location	Jumps to
Ø	$C6ØØ-C6FF	$Ø8ØØ-Ø8FF	$Ø8Ø1
1	$Ø8Ø1-Ø8FF	$B6ØØ-BFFF	$B7ØØ
2	$B6ØØ-BFFF	$9DØØ-B5FF	RUN

Now, in order to change the code in the boot so that it doesn't run away from us, we can either alter memory or alter (a copy of) the disk.

Check Routines

If you defeat the DOS (Disk Operating System) error-checking routines, you can copy the *Zaxxon* disk with *COPYA*. Of course, it won't run because of the 'check' routine and because of some of the other incompatibilities with normal *DOS 3.3*, but you can read and write to the copy easily.

To defeat the error-checking, enter the Monitor:

```
CALL-151
```

and type:

```
B942:18
```

Whenever DOS encounters an error, it jumps to a routine at $B942 which sets the carry bit and returns. The carry bit is a flag to DOS that there was an error and directs it to stop whatever it was doing. It then prints out a worthless message to the user.

If you defeat this routine, you can fool DOS and read the entire *Zaxxon* disk. Then copy the disk and examine the data on it. So, put your *DOS 3.3 System Master* in the drive and:

```
RUN COPYA
```

When the prompt for source drive is asked, hit:

```
CTRL C
```

This will break you into BASIC. Now, type the following:

```
CALL-151
3A1:18
302:17
35F:17
3D0G
70
RUN
```

This will make some changes to *COPYA.OBJ0* so that it ignores errors and only copies Tracks $0—$16. Line 70 of the Applesoft part of *COPYA* is also deleted to prevent loading in *COPYA.OBJ0* and writing over the change just made. Now, select the desired drives and copy *Zaxxon*.

Note: *Zaxxon* only lives at Track $0—$16 (Tracks $0—$13 on some versions). If *COPYA* tries to read an unformatted track your drive will recalibrate ('grind') for every sector that the program tries to read. Do not interrupt this process. After the copy is made, make another copy of the duplicate of *Zaxxon*, which you had just made. Label the first copy *WORK ZAX* and the second copy *COPYA ZAX*.

Put these two copies aside for a moment. The next step is to trace the boot. First, enter the Monitor:

```
CALL-151
```

and clear out memory with this command:

```
800:0 N 801<800.95FFM
```

To start the boot you need to run the code in the disk controller card, but we stop it before it runs away. You cannot change the code in ROM (Read Only Memory), but you can copy it down to

RAM (Random Access Memory) and change it. Use the following command from monitor to do this:

96ØØ<C6ØØ.C6FFM

Having done this, you can change the boot code so that it loads in Track $Ø, Sector $Ø, but does not execute it. At location $96F8 you will see a JMP $8Ø1. This starts the next boot process (see Boot Stages chart). Change this to JMP $FF59, which will jump to the Monitor. From the Monitor, type:

96F8:4C 59 FF

Now, put the *WORK ZAX* disk in drive one and execute the code with:

96ØØG

The drive will spin, the Apple will beep and you will see the asterisk (*) prompt.

From the table, you can see that this code is loaded into $8Ø1. (To stop the drive from spinning, enter **CØE8** RETURN from the Monitor. Also, to turn off the hi-res page, enter **CØ54** RETURN and **CØ51** RETURN). Now, examine the code at $8Ø1 with the command:

8Ø1L

Upon examining the code, you will find that it is fairly normal, with some exceptions:

```
Ø8Ø1-   A5 27      LDA $27        !-------------
Ø8Ø3-   C9 Ø9      CMP #$Ø9       !
Ø8Ø5-   DØ 18      BNE $Ø81F      !
Ø8Ø7-   A5 2B      LDA $2B        !      N
Ø8Ø9-   4A         LSR            !
Ø8ØA-   4A         LSR            !      O
Ø8ØB-   4A         LSR            !
Ø8ØC-   4A         LSR            !      R
Ø8ØD-   Ø9 CØ      ORA #$CØ       !
Ø8ØF-   85 3F      STA $3F        !      M
Ø811-   A9 5C      LDA #$5C       !
Ø813-   85 3E      STA $3E        !      A
Ø815-   18         CLC            !
Ø816-   AD FE Ø8   LDA $Ø8FE      !      L
Ø819-   6D FF Ø8   ADC $Ø8FF      !
Ø81C-   8D FE Ø8   STA $Ø8FE      !
Ø81F-   AE FF Ø8   LDX $Ø8FF      !
Ø822-   3Ø 15      BMI $Ø839      !      D
Ø824-   BD 4D Ø8   LDA $Ø84D,X    !
Ø827-   85 3D      STA $3D        !      O
Ø829-   CE FF Ø8   DEC $Ø8FF      !
Ø82C-   AD FE Ø8   LDA $Ø8FE      !      S
```

```
Ø82F-    85 27        STA $27        !
Ø831-    CE FE Ø8     DEC $Ø8FE      !
Ø834-    A6 2B        LDX 2B         !
Ø836-    6C 3E ØØ     JMP ($ØØ3E)    !
Ø839-    EE FE Ø8     INC $Ø8FE      !-------------
Ø83C-    AD 55 CØ     LDA $CØ55      !   TURN ON
Ø83F-    AD 5Ø CØ     LDA $CØ5Ø      !   HI-RES 2
Ø842-    AD 57 CØ     LDA $CØ57      !-------------
Ø845-    A2 7D        LDX #$7D       !     see
Ø847-    9A           TXS            !   article
Ø848-    4C B4 Ø8     JMP $Ø8B4      !-------------
         :  :  :                     !
         :  :  :                     !
         :  :  :                     !
Ø8B4-    A9 2Ø        LDA #$2Ø       !
Ø8B6-    85 1B        STA $1B        !
Ø8B8-    AD 52 CØ     LDA $CØ52      !
Ø8BB-    A9 6Ø        LDA #$6Ø       !
Ø8BD-    8D Ø6 Ø7     STA $7Ø6       !-------------
Ø8CØ-    6C F8 Ø8     JMP ($Ø8FD)    ! JUMP TO $8ØØØ
```

How it Works

The first part of the code, at $8Ø1—$83B, is lifted verbatim from a *DOS 3.3* Slave disk. This code loads in Track $Ø, Sector $Ø, through Track $Ø, Sector $Ø9, into $7FØØ—$88FF. This is revealed from location $8FE, which is one higher than the first page loaded into. The byte at $8FF is one less than the number of sectors to be loaded.

The next piece of code turns on the hi-res screen. The last part of code, before the jump to $8B4, looks innocent, but really isn't. It loads the X-register with $7D and transfers it to the stack pointer.

To understand the implications of this, you must understand how the computer keeps track of its return position after an RTS (Return from Sub-routine).

When the 65Ø2 encounters a JSR (Jump Sub-Routine), it stores the present address on the stack so it knows where to return to when an RTS is encountered.

This can be used to obscure code from unwanted eyes. For example, say we want to go to $96ØØ. You can load the stack with $95 and then $FF, by using the PHA op-code (PusH Accumulator on Stack). When an RTS is encountered, the two last bytes are pulled from the stack, incremented by one (to $96ØØ), and jumped to.

Alternatively, you can change where the pointer on the stack is pointing to and make it point to the desired location. Keep this in mind when you find an RTS.

At $8B4, a few zero page locations are loaded and then there is an indirect jump to $8ØØØ (through $8FD). To see the code at $8ØØØ load the next stage of the boot but stop it before it can execute. To do this, run your sector editor and change Track $Ø, Sector $Ø, byte $CØ, to 4C 59 FF on the disk labeled *WORK ZAX*.

This will jump you to the monitor before it can execute the code at $8ØØØ. Now, write the sector back out to the disk and boot the disk. After a moment the computer will beep and stop in the Monitor. You can now examine the code at $8ØØØ. Do this with the command:

8ØØØL

and the code will look like this:

```
8ØØØ-   AØ Ø9        LDY #$Ø9        !-------------
8ØØ2-   A2 ØØ        LDX #$ØØ        !
8ØØ4-   8A           TXA             !   M
8ØØ5-   EE ØD 8Ø     INC $8ØØD       !   E     M
8ØØ8-   EE 1Ø 8Ø     INC $8Ø1Ø       !   M     O
8ØØB-   BD 18 7F     LDA $7F18,X     !   O     V
8ØØE-   9D 7E ØØ     STA $ØØ7E,X     !   R     E
8Ø11-   CA           DEX             !   Y
8Ø12-   DØ F9        BNE $8ØØB       !
8Ø14-   88           DEY             !
8Ø15-   DO EE        BNE $8ØØ5       !-------------
8Ø17-   6Ø           RTS             !JUMP THRU STACK
                                     !-------------
```

Notice the RTS at $8Ø17. Remember: you jumped (JMP), not jumped sub-routine (JSR), to all the routines, so there is nothing on the stack to return to!

Well, yes there is! The memory move relocates the memory at $7F19 through $8718, down to $7F through $Ø87E. This moves memory across page $1, which is the stack!

Remember too, that the stack pointer is set to $7D in Boot1. After the memory move, an RTS is executed. The stack is pointing at $17D, which is now $Ø7 and $65 after the memory move. This will be the next jump (plus one) for the final stage of the boot!

The manufacturer, Datasoft, has added a final bit of protection in that the next jump is across the text page which, of course, changes when you exit the program in any manner. But you can simply move memory to, say, $1Ø7E and examine it to see what the next load does.

To do this, change $8Ø1Ø to 1Ø and execute the memory move. Do this as follows:

8Ø1Ø:1Ø N 8ØØØG

Now, $1766 is equivalent to $766. So type:

1766L

and examine the code. It should look as follows:

```
1766-   A5 2B          LDA $2B         !
1768-   8D ØE Ø2       STA $Ø2ØE       !   G    C
176B-   A9 2Ø          LDA #$2Ø        !   O    O
176D-   EA             NOP             !   O    D
176E-   A6 2B          LDX $2B         !   D    E
177Ø-   2Ø 1F Ø2       JSR $Ø21F       !------------
1773-   AØ 2Ø          LDY #$2Ø        !
         :  :  :                       !  NIBBLE
         :  :  :                       !  COUNT/CHECK
         :  :  :                       !
         :  :  :                       !------------
17D4-   A9 16          LDA #$16        !   G    C
17D6-   8D 11 Ø2       STA $Ø211       !   O    O
17D9-   A9 DØ          LDA #$DØ        !   O    D
17DB-   8D 16 Ø2       STA $Ø216       !   D    E
17DE-   4C 9A Ø1       JMP $Ø19A       !------------
```

If you take time to look at this code you will find that the offending 'nibble count' or 'checksum' starts at $1773 and goes to $17D3! This is all you need to know to defeat it.

What you have to do, next, is jump around this. Do it by typing:

1773:4C D4 Ø7

The other loads and stores of the accumulator are parameters for their loader. For further understanding, here are the parameters:

$211 - high track # to start from.
$212 - high sector # to start from.
$21E - # of pages (sectors) to load.
$21B - starting page to load at.
JSR $1E9 - start the load.

The last thing to do is to find the 'nibble count' code on the disk code and change it. Most good sector editors (like *The Inspector*) have LOCATE routines enabling you to find a pattern of bytes on a disk.

One final note: Now that you know where the check routine lives, we can defeat it. But you also must change the epilog bytes to normal *DOS 3.3*.

Datasoft uses $CC for their epilogue bytes where normal DOS uses $DE. If you are familiar with loaders and RWTS, you can find it in their loader. The table is on page $4 (page $14).

Stepwise

With this information in hand, here is a step-by-step procedure for the deprotection of *Zaxxon* (*Mockingboard* version only):

1 Get out your *DOS 3.3 System Master* disk and run it:

```
RUN COPYA
```

2 When *COPYA* asks for the slot number of the original disk, stop the program:

```
CTRL C
```

3 Enter the Monitor:

```
CALL -151
```

4 Patch DOS and *COPY.OBJØ* so that they ignore read errors and only Tracks $Ø—$16 are copied

```
B942:18
3A1:18
3Ø2:17
35F:17
```

5 Re-enter Applesoft and delete Line 7Ø of *COPYA* so that *COPY.OBJØ* is not reloaded

```
3DØG
7Ø
```

6 Run *COPYA* to copy *Zaxxon*:

```
RUN
```

7 If you have two disk drives you can leave the room during the copy process; otherwise, you may have to put up with your drive making some horrible grinding noises. *Don't worry. This will not harm the disks or your drives.*

8 Run your sector editor and make the following changes depending on which version of *Zaxxon* you have. (Note: you may have to try all of these changes, depending on the release date of your Zaxxon. One of them should work, though):

For Mockingboard Versions

Track	Sector	Byte	From	To
$ØØ	$Ø4	$4F	$CC	$DE
$ØØ	$Ø4	$5Ø	$DØ	$EA
$ØØ	$Ø4	$51	$AE	$EA
$ØØ	$Ø7	$ØD	$AØ	$4C
$ØØ	$Ø7	$ØE	$2Ø	$D4
$ØØ	$Ø7	$ØF	$84	$Ø7

For Mockingboard Versions for which the first method does not work

Track	Sector	Byte	From	To
$00	$07	$00	$A9	$4C
$00	$07	$01	$01	$C0
$00	$07	$02	$48	$08
$00	$04	$4F	$CC	$DE

For the older non-Mockingboard Versions

Track	Sector	Byte	From	To
$00	$07	$1F	$A9	$4C
$00	$07	$20	$00	$C0
$00	$07	$21	$85	$08
$00	$04	$4F	$CC	$DE
$00	$04	$50	$D0	$EA
$00	$04	$51	$AE	$EA

9 Don't forget to write the sectors you have altered back to the deprotected *Zaxxon* disk.

Clay Harrel's APT for...

Zaxxon

More Planes

The *Zaxxon* in the arcades gives you five planes and our *Zaxxon* only three! If you want more planes, change Byte $17 on Track $09, Sector $08 with your sector editor to the number of planes you want (between $00 and $FF). I chose to change this byte to $03. This gave me four planes, which is a nice compromise. This modification applies to all versions of *Zaxxon*.

Hayden Software

Hayden

Softkey For Hayden Software
by Floyd Splidnik
(Hardcore COMPUTIST # 8, page 6)

Requirements:

Apple][Plus with 48K
Integer card or some means to RESET into the Monitor
One disk drive with *DOS 3.3*
DEMUFFIN PLUS (see the **How To Make DEMUFFIN PLUS** article)
A *DOS 3.3* slave disk with a null HELLO program
Any of several original program disks from Hayden

I have found that the following procedure will work on a number of releases from Hayden such as *Shuttle Intercept, Alibi, Kamikaze*, etc. Unfortunately, this procedure will NOT work on *Sargon II* or *Sargon III.*

The technique works because the Hayden disks I mentioned use a 13-sector DOS that has been modified but still has the normal routines intact and in their proper locations. *DEMUFFIN PLUS* is used to read the original disk with the Hayden DOS and then write with normal *3.3 DOS*.

If you don't already have DEMUFFIN PLUS, you'll have to create it following the instructions in the **How To Make DEMUFFIN PLUS** *article in this Book Of Softkeys.*

1 INITialize one or more disks with a 'null' *HELLO* program:

```
FP
INIT HELLO
```

2 Boot up with one of the Hayden Disks:

```
PR#6
```

3 As soon as Applesoft prompt "] " appears on the screen, hit RESET to enter the Monitor.

4 To test if this technique will work on your disk, do a call to the CATALOG entry point:

```
A56EG
```

If a CATALOG is displayed, then this technique should work for the disk in question.

5 Now move the Hayden RWTS out of the way so it is not destroyed when normal DOS is booted:

```
6000<B800.BFFFM
```

6 Next, boot up with one of the disks INITialized in step 1:

```
6 CTRL P
```

7 Load, but do not run, *DEMUFFIN PLUS*:

```
BLOAD DEMUFFIN PLUS
```

8 Enter the Monitor:

```
CALL -151
```

9 Move the Hayden RWTS back into position so that *DEMUFFIN PLUS* can utilize it:

```
B800<6000.67FFM
```

10 Start execution of *DEMUFFIN PLUS*:

```
803G
```

11 Transfer all the files from the original disk to an initialized 3.3 disk using the wildcard character (=). The HELLO file on the 3.3 disk should be replaced with the HELLO file from the Hayden disk.

The above technique not only works on the Hayden disks but also on a number of other disks which display an Integer (<) prompt when booting. It is not always quite so easy as with the Hayden disks however, as nibble counts and/or other protection sometimes need to be eliminated.

Sierra On-Line
Software**

Softkey For Sierra On-Line Software
by Doni G. Grande & Clay Harrell
(Hardcore COMPUTIST # 9, page 24)

*****Includes the following software packages:***
Screenwriter II V2.2
The Dic *tion *ary
Sammy Lightfoot
Time Zone V1.1
Apple Cider Spider
Oil's Well
Cannonball Blitz

Requirements:
Apple with 48K
COPYA
A Sector editing program such as *Disk Zap*
A blank disk for each of the above programs

The folks at Sierra On-Line have come to a very realistic view on copy-protection. Some of their earlier releases such as *Lunar Leepers* used elaborate protection schemes like spiral tracking (uses ¼ tracks spiraled along the disk). The problem with this kind of protection is that it is expensive, and it doesn't work on all flavors of Apples (//e, etc.). On-Line probably also had to send out to have the copies made because you can't exactly run *COPYA* to make thousands of commercial copies of a spiral disk.

With competition increasing, and the high awareness to cut internal costs, On-Line has chosen not to use elaborate protection schemes on their newer releases (*Oil's Well, Sammy Lightfoot, Screenwriter II, ver 2.2*, etc.). Instead, they use a good basic scheme that will often deter those equipped with *Locksmith* or *Nibbles Away*, and even discourage those who try to deprotect their programs altogether. This protection scheme is interesting in that one can usually use *COPYA* to copy the disk, but the copy will not run. Somehow, the programs know that an original disk is not in the drive.

Counting Nibbles

The basis of Sierra On-Line's protection scheme is called a 'nibble count'. When the master disk is copied at On-line, the copy program counts the number of bytes on a certain track and then stores this value on the disk. This value is different for each copy of the program because miniscule changes in the disk drive speed can cause extra or fewer bytes to be put on a track when it is written. Normal DOS does not care about this, since it uses a certain byte (**FF**) as a filler. When a disk is booted, it reads that track and compares the number of bytes read to the number stored on the disk. If they don't match, the program bombs out! *Locksmith* and several other bit copy programs allow you to do nibble counting when making a copy. However, if you have ever attempted this, you know how very difficult it is and how long it takes to get a reliable copy. Though the disk drive speed is highly regulated, it only takes a fraction of a turn to måke a difference of a few bytes on the track.

The best way around the problem is to find the protection code and disable it. Fortunately for us, Sierra On-line uses the same technique on a number of their disks, so if you figure out how to copy one, you can copy a number of them. If you are interested in how they implement nibble counting, read on. Otherwise, skip the next section and go on to the **The Steps**.

Real Men Write Self-Modifying Code

The protection code used by Sierra On-line is a great example of hiding the true function of machine code! Read on, and you will see examples of code within code within code. A typical disassembly of the protection code might start out:

```
0900  CE 03 09   DEC $0903
0903  EF         ???
0904  03         ???
0905  09 AD 28   ORA #$AD
0907  28         PLP
0908  09 49      ORA #$49
```

It looks like there really isn't anything there. But notice that the first instruction decrements the very next byte! If we look at this same code after the first statement executes, we see:

```
0900-  CE 03 09   DEC $0903
0903-  EE 03 09   INC $0903
0906-  AD 28 09   LDA $0928
0909-  49 8A      EOR #$8A
090B-  D0 01      BNE $090E
090D-  20 8D 28   JSR $288D
0910-  09 18      ORA #$18
```

It doesn't even look like the same code! Notice, too, that the 'new' instruction at $Ø9Ø3 restores $Ø9Ø3 to its original value of $EF so that the program hides itself again.

From there, a simple sequence of instructions first gets a byte from $Ø928 (which happens to be $6Ø), EORs it with $8A (giving $EA) and then executes a branch instruction.

As I have shown, the accumulator is not zero, so this branch is taken...*right into the middle of another statement!* $Ø9ØE is in the middle of the following JSR! However, by listing from $Ø9ØE, the following code is revealed:

```
Ø9ØE-  8D 28 Ø9    STA $Ø928
Ø911-  18          CLC
Ø912-  DØ Ø1       BNE $Ø915
Ø914-  4C AØ 29    JMP $29AØ
Ø917-  98          TYA
```

Ah ha! There is really a STA hiding there, and it stores the accumulator back to $Ø928. Then a Clear Carry is done to set up the carry flag to be used in a moment. The branch is always done since the accumulator doesn't end up being zero (it is $EA, from above). Once again, this branch is to the middle of another instruction! Listing from $Ø915, you see the following:

```
Ø915-  AØ 29       LDY #$29
Ø917-  98          TYA
Ø918-  9Ø Ø1       BCC $Ø91B
Ø91A-  2Ø 59 ØØ    JSR $ØØ59
Ø91D-  Ø9 99       ORA #$99
```

So, another bit of hidden code! The Y-register is loaded with the value $29, copied into the accumulator, and a branch is made (remember the carry flag that was cleared above?).

Again, the branch is to the middle of another instruction (are you beginning to get the picture?). Disassembling the program from $Ø91B gives:

```
Ø91B-  59 ØØ Ø9    EOR $Ø9ØØ,Y
Ø91E-  99 ØØ Ø9    STA $Ø9ØØ,Y
Ø921-  C8          INY
Ø922-  DØ F3       BNE $Ø917
Ø924-  88          DEY
Ø925-  3Ø Ø1       BMI $Ø928
Ø927-  4C EA E1    JMP $E1EA
```

Here at last is the core of the code (no pun intended). This section does an Exclusive OR of a memory location with the accumulator and then stores the number back into memory.

The Y register (used as the index) is incremented and the branch to the top of the loop is taken until Y is zero (it counts from $29 to $FF). When the Y-register is zero it is decremented, and since it is now negative, the branch is made into the middle of another instruction. You notice that this is at $Ø928, our old friend from above which was converted from a $6Ø (RTS) to an $EA (NOP).

Once all the EOR's have been executed on memory from $Ø929—$Ø9FF, the following code is revealed.

```
Ø928-   EA         NOP
Ø929-   C8         INY
Ø92A-   8C F4 B7   STY $B7F4
Ø92D-   8D EC B7   STA $B7EC
Ø93Ø-   A9 B7      LDA #$B7
Ø932-   AØ E8      LDY #$E8
Ø934-   2Ø D9 Ø3   JSR $Ø3D9
Ø937-   BD 89 CØ   LDA $CØ89,X
Ø93A-   A9 Ø5      LDA #$Ø5
Ø93C-   8D ØØ BB   STA $BBØØ
Ø93F-   2Ø 9Ø Ø9   JSR $Ø99Ø
Ø942-   1Ø Ø1      BPL $Ø945
Ø944-   2Ø C8 CØ   JSR $CØC8
Ø947-   3Ø 5D      BMI $Ø9A6
Ø949-   8C CØ 9Ø   STY $9ØCØ
Ø94C-   F8         SED
Ø94D-   BD 8C CØ   LDA $CØ8C,X
Ø95Ø-   1Ø ØA      BPL $Ø95C
Ø952-   C9 C9      CMP #$C9
Ø954-   DØ ØD      BNE $Ø963
Ø956-   BD 88 CØ   LDA $CØ88,X
Ø959-   4C ØØ Ø9   JMP $Ø9ØØ
```

This is the start of On-line's infamous 'nibble count' routine which reads Track Ø and counts the bytes on the track. If the count is not correct, then the program proceeds to wipe out the computer's memory. If the count is correct, a jump is made to $Ø9ØØ which is the start of the entire routine. The EOR operations are performed all over again and the result is that the valid code is returned to its original, obscured state. Pretty sly those On-line people!

How does the routine ever stop? Remember that the program first EORed location $Ø928 ($6Ø) with $8A and then stored the result ($EA) back into $Ø928, which was later branched to? The second time the program is executed, the contents of $Ø928 ($EA) are EORed with $8A, giving a value $6Ø (the original value) which is then put back into $Ø928. When the branch to $Ø928 is later executed, it encounters $6Ø, which just happens to be the machine code for RTS!

As you can see, the people who write the protection schemes are clever. However, sometimes they are so clever that they fool themselves, and that is their downfall. Apparently, they believed that the above code was so clever, and that it was so well hidden that they relied totally on IT protecting their software. In other words, it is the only check done to see if the disk in the drive is an original. Also, probably since Sierra On-line publishes software written by a number of authors, this protection program is called as a *stand-alone subroutine*, returning no values! The result is that it can usually be easily defeated just by changing the first byte to $60 to keep it from executing at all!

I mentioned above that usually the protection scheme could be defeated by changing the first byte of the protection code to $60. One case where this won't work is in *Time Zone*. *Time Zone* does a checksum on the protection code several times before it actually executes the protection code (*Sammy Lightfoot* also does a checksum between loading each level after the first!). The checksum code for *Time Zone*, whose nibble count routine lives at $1700, looks something like the following:

```
-----------------------------------Initalize Pointer to $1700
     50AC-  A9 00     LDA #$00
     50AE-  85 FE     STA $FE
     50B0-  A9 17     LDA #$17
     50B2-  85 FF     STA $FF
------------------------------------Initialize all registers
     50B4-  A9 00     LDA #$00
     50B6-  A0 00     LDY #$00
     50B8-  A2 00     LDX #$00
-----------------------------------------Ready Carry for ADD
     50BA-  18        CLC
--------------------------------------------Add memory byte
     50BB-  71 FE     ADC ($FE),Y
------------------------------------------------Multiply by 2
     50BD-  0A        ASL
--------------------------------------------Increment index
     50BE-  C8        INY
-------------------------------Decrement counter (256 times)
     50BF   CA        DEX
--------------------------------------------Loop until done
     50C0-  D0 F9     BNE $50BB;
-------------------------------------Compare Checksum to $1E
     50C2-  C9 1E     CMP #$1E;
     50C4-  EA        NOP
--------------------------------------------Branch if equal
     50C5-  F0 03     BEQ $50CA
-------------------------------------$09CB contains JMP $09CB
     50C7-  4C CB 09  JMP $09CB
------------------------------------Return to calling routine
     50CA-  60        RTS
```

The above assumes the protection code is located at $1700, which it is in *Time Zone*. The easiest way to foil this checksum routine is to change the first byte of the nibble count routine to a $60 (RTS) and also to alter the next byte of the routine so that the checksums will be identical even though the code has been changed. By experimenting with the code listed above, it turns out that the second byte of the nibble count routine should be changed to a $E0. However, this will only work on *Time Zone* because the other programs seem to use a different checksum routine. For these other programs (*Oils Well, Sammy Lightfoot*, etc.) it turns out that the second byte needs to be changed to a $AD. *Screenwriter II ver. 2.2* and *The Dic*tion*ary* do not have the checksum routines, so you can get away with just changing the first byte of the nibble count routine to a $60.

The Steps

In a step-by-step fashion, here is what you need to do to make a copyable version of many of Sierra On-line's programs:

1 *COPYA* the original disk:

```
RUN COPYA
```

2 Run your sector editor. Examine each sector for the values $CE $03. If you have an editor such as *Disk Zap* from *Bag of Tricks*, or *Tricky Dick* with *The Tracer* you can automatically search for this sequence. I have found that this sequence of bytes is usually at the very beginning of a sector.

Note: *Sometimes this code appears more than once, as in Screenwriter II, so be sure to search the entire disk to find all occurences!*

3 Change the $CE to $60 and the $03 to an $AD ($E0 for *Time Zone*) and rewrite the sector, and that's all there is to it!

Screenwriter II ver. 2.2	Track	Sector	Byte	From	To
	1A	0E	00	CE	60
	08	0F	00	CE	60
	0C	0F	00	CE	60
	17	0F	00	CE	60

Dic*tion*ary	Track	Sector	Byte	From	To
	10	0D	00	CE	60

Sammy Lightfoot	Track	Sector	Byte	From	To
	05	0E	00	CE	60
	05	0E	01	03	AD

Time Zone ver. 1.1	Track	Sector	Byte	From	To
	03	0F	00	CE	60
	03	0F	01	03	E0

Apple Cider Spider	Track	Sector	Byte	From	To
	12	01	00	CE	60
	12	01	01	03	AD

Oil's Well

Track	Sector	Byte	From	To
10	0F	00	CE	60
10	0F	01	03	AD

Cannonball Blitz

Track	Sector	Byte	From	To
18	06	00	CE	60
18	06	00	03	AD

An Alternate Method

An alternate method which worked for *Sammy Lightfoot* and *Time Zone* involves disabling the code which does a JSR to the nibble count routine. This technique may also work on other SOL disks which directly JSR or JMP to the nibble count routine. *Apple Cider Spider* and *Oil's Well* apparently do an indirect JMP or JSR to their nibble count routines so this alternate technique will not work on them.

1 *COPYA* the disk:

```
RUN COPYA
```

2 Use a sector editor to search the disk for a byte sequence of $CE $03 which is usually found at the beginning of a sector.

3 When you find this sequence, look at the byte which lies eight bytes past the $CE. This byte is the high order byte of the address where the nibble count routine runs.

Write down the value of this byte. For example, if the $CE is found at the beginning of the sector and the eighth byte is 09, then the protection code is located at $0900.

Now search the disk for a JSR $0900 (20 00 09). This will be the call to the protection code.

Note: *There may be more than one call to this code, so be sure to search the entire disk!*

4 Change the JSR XXXX you find to EA EA EA, rewrite the sector, and you are done!

For *Time Zone* and *Sammy Lightfoot*, here are the sector edits which are needed.

Sammy Lightfoot

Track	Sector	Byte	From	To
0D	00	9B	20	EA
0D	00	9C	00	EA
0D	00	9D	9E	EA

Time Zone ver. 1.1	Track	Sector	Byte	From	To
	03	0B	F0	20	EA
	03	0B	F1	00	EA
	03	0B	F2	17	EA

Hint: If you try the first method and everything seems to work fine up to a point and then the program just hangs, try the alternate method.

Screenwriter 2.2 update

by Andrew Reiffenstein

The article on Softkey For Online Software did not have a complete softkey for Screenwriter version 2.2 as it only deprotects the RUNOFF (Printout) and not the EDITOR. The complete softkey is as follows:

Using a disk editor (*Disk Fixer 3.3*,*Zap*,etc), read in Track $0E, Sector $03. Locate the sequence **20 00 6E** (byte offset of rest of 49). Change it to **EA EA EA**. The EDITOR will now work.

The rest of the softkey was correct as published, that is: read in Track $0F, Sector $07. Locate the sequence **20 00 7F** (byte offset of 90). Change it to **EA EA EA**. The RUNOFF part will now work.

Plese note that it is advisable to make these changes on a COPYAed copy of the original Screenwriter 2.2 disk. You should NEVER change the original.

How To Make...

DEMUFFIN PLUS

(Hardcore COMPUTIST # 8, page 15)

1 Boot *DOS 3.3 Master* to load Integer BASIC.

2 Enter the other BASIC, load MUFFIN, then enter the Monitor:

```
INT
BLOAD MUFFIN
CALL -151
```

3 Initialize the 'Programmer's Aid' relocation feature:

```
D4D5G
```

4 Tell the Monitor what is being moved and where it is going:

```
1900<B800.BFFF CTRL Y *
```

5 Relocate the first chunk of code:

```
1900<B800.BA10 CTRL Y *
```

6 Move the data segment:

```
.BC57M
```

7 Relocate the rest of the code:

```
.BFFF CTRL Y
```

8 Make the following modifications to MUFFIN and the relocated RWTS subroutine:

```
1155:00 1E
115B:D9 03
1197:A0 20
15A0:A0 D2 C5 D3 C9 C4 C5 CE
15A8:D4 A0 C4 AE CF AE D3 AE
15F7:C4 C5
20A0:A9 1E 8D B9 B7 20 FD AA
20A8:48 A9 BD 8D B9 B7 68 60
```

9 Save this new code:

```
BSAVE DEMUFFIN PLUS,A$803,L$1900
```

Instructions on its use can be found in the following softkey articles:

Data Factory 5.0
Hayden Software

Super I.O.B.

version 1.5

by Ray Darrah

Requirements:
Apple][Plus, //e or //c
One *DOS 3.3* disk drive

Somewhere near the beginning of time, as Hardcore COMPUTIST knows it, an Applesoft program called *IOB* was written by 'Bobby.' This program was designed to deprotect disks by interfacing directly with a machine language program in DOS called the **Read/Write Track Sector** program (RWTS). *IOB* was so named because it modified a parameter list in DOS called the **Input/Output control Block** (IOB). At first, quite a bit of confusion was generated due to the fact that the Applesoft program had the same name as the parameter list used by the RWTS.

Not quite as long ago (a few short eons in the evolution of Hardcore COMPUTIST) a program with the same concept as *IOB* was introduced. This new program was called Super IOB because of the several advantages it offered over the original *IOB*. Perhaps the most notable of these is the controller concept in which a short subprogram is keyed into the main portion of the Super IOB program thus changing it adequately to allow it to copy many disks with varied protection schemes. Although *IOB* incorporated this same concept, due to the structure of the program, it was often necessary to change other lines of the program in addition to keying in a controller.

When you get right down to it, Super IOB's power is derived primarily from a small collection of Applesoft subroutines. These subroutines are called by the inserted subprogram (the Super IOB controller) in a particular sequence to allow deprotection of individual programs. The concept of Super IOB is simple, but only by understanding the function of the subroutines in Super IOB can an Applesoft programmer learn to create a controller for a specific disk.

Since the introduction of Super IOB, Hardcore COMPUTIST has used this flexible program to deprotect (or partially deprotect) dozens of commercial programs. In addition, Hardcore COMPUTIST has printed several utilities designed to aid the programmer in his use of Super IOB. These included 'CSaver' (Hardcore COMPUTIST # 13), a machine language program which helps in the merging of the controller.

Presented here is the second update to Super IOB. Because of an addition to the machine language portion of Super IOB and the inclusion of a new subroutine this new version of Super IOB now can copy a disk up to 350% faster than its predecessors.

Keying in Super IOB for the First Time

If you don't already have Super IOB, you will have to use the procedures outlined in the **What Is A Softkey** article to key in the Applesoft and machine language portions of the program. Save the Applesoft portion with:

SAVE SUPER IOB RETURN

Save the new machine language portion with:

BSAVE IOB.OBJ0, A$300, L$A8 RETURN

The Function Of *Super IOB*

Super IOB de-protects disks by pushing the RWTS subroutine in DOS to its upper-most limits. Because of this, it only copies disks with sectors that somewhat resemble normal sectors.

Before a disk can be softkeyed, the sector alterations must be determined. The easiest way to do this is to use a nibble editor program (like *Bag of Tricks*, or *The Nibbler* or *CIA's Linguist*) which shows these sector differences.

Once the protection has been discovered, a controller program (Lines 1000—9999) designed to deprotect such a scheme must be inserted into Super IOB. Here is a list of the protection schemes Super IOB was designed to softkey:

1) Altered address, data, prologue, or epilogue marks
2) Strangely numbered sectors or tracks
3) Modified RWTS (with same entry conditions)
4) Half tracks for any of the above
5) 13- or 16-sector base format for any of the above

A Little Briefing

The following is a short discussion of the protection scheme and how each relates to Super IOB. Keep in mind that often more than one scheme is used at a time. This has the effect of complicating the Super IOB controller.

Altered Marks

DOS looks for specific marks when trying to read a sector. Changing these is a common practice, especially on older releases. DOS puts certain reserved bytes on the disk (during INITialization) so it can determine where a sector begins and ends.

For example, a normal 16-sector disk has the bytes D5 AA AD designating the **Start of the Data Field** which contains the 256 bytes of a sector in encoded form. When a standard RWTS tries to find a sector, it looks for these marks. If they are not found, (either because they don't exist or they have been changed to something else) DOS returns with the dreaded *I/O ERROR*.

The sequences of the four reserved-byte marks (**Start of Address, End of Address, Start of Data, End of Data**) are handled by subroutines in Super IOB. These subroutines change the marks that the current RWTS looks for when reading.

Strangely Numbered Sectors

Within an address field, there are 8 bytes which alert the RWTS to which sector is about to pass under the read/write head. On some disks, these are not standard. These disks are easily softkeyed with Super IOB. The controller instructs Super IOB to read the sectors using the strange sector numbers and then write them using the correct numbers. This works because the RWTS merely compares the sector number found on the disk with the one the controller is looking for (even if it is higher than 15 and therefore illegal).

Modified RWTS

The disk-protectors will often rearrange and/or modify the standard RWTS subroutine. If this is the case, you must first save the strange RWTS onto a normal DOS disk and then use a controller which reads the protected disk using the strange RWTS and then writes via the normal 3.3 RWTS.

Such a controller is included in this article. It is called the **Swap controller**. This is because of its use of the 'Swap RWTS at $1900 with the one at $B800' routine in Super IOB.

Since the RWTS of a protected disk will be modified to read any altered DOS marks, this is an easy method to use if you are unable to determine the protection scheme.

The Stepper Motor

There is a motor inside your disk drive that is responsible for moving the disk head to the various tracks.

Known as the 'Stepper Motor', it has **four** electromagnets (numbered Ø to 3) that can be turned 'on' or 'off' by referencing memory locations. When one of these magnets is turned 'on', the permanent magnets in the motor are attracted to it and the motor shaft turns until the permanent magnets are aligned with the electromagnets.

To obtain continuous motion, a program would:

1 Turn a magnet (called phase) 'on.'

2 Wait for the motor to get aligned (it doesn't take much time).

3 Turn 'off' the magnet.

4 Turn 'on' the next adjacent magnet (the next magnet differs depending on whether you want to go to a higher or lower track).

5 Go to Step 2.

Odd Phases = Half Tracks

Because of the resolution of the disk head combined with the accuracy of the stepper motor, normal DOS tracks are placed only on the **even** phases.

This means that for every track DOS moves, it references two magnets. As a result, the disk head never stops at any of the odd phases (i.e. aligned with magnets 1 or 3).

Therefore, the odd phases are commonly called **half-tracks**. The disk-protectors will often put information on these phases that are inaccessible to normal DOS.

A routine called 'Move S Phases' (in Super IOB) handles the job of getting to these so-called half-tracks and can also be used (by a controller) to get to tracks that have been marked as other tracks.

A complete discussion of how to use this routine appears later in this article.

Anatomy Of A Controller

Before we attempt to write a controller, let's look at the subroutines at the controller's disposal. During this explanation, it would be wise to refer to the **Super IOB Applesoft BASIC listing** of Super IOB to see how each is accomplished.

The New Routine

There has only been one routine added to Super IOB to arrive at Version 1.5. The explanation of its workings follows.

Name: **R/W A RANGE**
Line Number(s): **61Ø — 62Ø**
Entry Conditions:

TK, ST = first sector to read or write
LT, LS = last sector (minus one MOD 16) to read or write
MB = maximum buffer page
CD = command code

Function: Quickly reads or writes a range of sectors by calling a machine language program. To be faster, this subroutine stores the sectors in memory in a **decreasing** manner. That is: Sector $ØF is followed by Sector $ØE and Sector $ØD and so on.

Storing the sectors in memory in a different sequence than ascending is O.K. as long as **1)** the sectors are written in the same order they are read, and **2)** 'The Sector Editor' routine knows that the sectors were stored in descending order by setting the variable **FAST** equal to **1**.

The Older Routines

The following is an explanation of the remainder of the routines found in Super IOB Version 1.5. These routines were included in the older Super IOB programs as well.

Name: **START UP**
Line Number(s): **1Ø to 6Ø**
Entry Conditions: Not Applicable

Function: The first few lines merely identify the program; however, Line 6Ø sets **HIMEM** and **LOMEM** so that they fit the memory usage requirements (see 'Memory Allocation Table'). It then goes to 'CONFIGURATION TIME'.

Name: **INITIAL IOB SETUP**
Line Number(s): **8Ø**
Entry Conditions:

DV = drive to be accessed
VL = volume of disk to be accessed
SO = slot to be accessed

Function: Normally GOSUBed via 'TOGGLE READ/ WRITE'. Its purpose is to reset the buffer page and set the drive, slot and volume number to the disk to be accessed next.

Name: **R/W SECTOR**
Line Number(s): **1ØØ — 11Ø**
Entry Conditions:

- **TK** = Track to be accessed
- **ST** = Sector to be accessed
- **CD** = Command code for the RWTS

Function: GOSUBed directly from the controller. It reads or writes (depending upon **CD**) at the specified track and sector.

Name: **MOVE S PHASES**
Line Number(s): **13Ø — 14Ø**
Entry Conditions:

- **SO** = Slot of drive to move
- **DV** = Drive number of drive to move
- **PH** = Phase number that the disk head is currently over
- **S** = Number of phases to move

Function: Moves the disk Read Head the number of phases specified by **S** (one phase equals one half-track) and is capable of moving in either direction up to 128 phases (or 64 tracks). Care should be taken that **PH** + **S** isn't greater than 255 or less than **Ø** or an error will occur.

Name: **ALTERED ENDING MARKS**
Line Number(s): **17Ø**
Entry Conditions: Proper **DATA** pointers
Function: Changes the Address Field and Data Field Epilogue markers in the normal RWTS. The values to change these to should be contained in a DATA statement. Because normal DOS only checks the first two bytes of these markers, only four values are required. The Address Field is changed first and should appear first in the DATA statement.

Name: **ALTERED ADDRESS MARKS**
Line Number(s): **19Ø**
Entry Conditions: Proper DATA pointers
Function: This routine modifies the RWTS (via POKE) so that it looks for a different sequence of Address Field Prologue marks. The decimal values of the marks to look for should be stored as the next DATA elements.

Name: **ALTERED DATA MARKS**
Line Number(s): **21Ø**
Entry Conditions: Proper DATA pointers
Function: Same as previous subroutine except for Data Field Prologue marks.

Name: **NORMALIZER**
Line Number(s): **230 — 250**
Entry Conditions: None

Function: Restores the values in the RWTS subroutine that are changed by any routine in Super IOB. This routine should be called just before writing in order to fix the RWTS so that it can access normal DOS disks.

Name: **IGNORE ADDRESS CHECKSUM**
Line Number(s): **270**
Entry Conditions: none

Function: Modifies the RWTS subroutine so that it doesn't examine the checksum byte of the address field. This routine has been incorporated in many previous controller.

Name: **ALTERED DATA CHECKSUM**
Line Number(s): **290**
Entry Conditions: Proper DATA pointers

Function: Alters the starting checksum byte that the RWTS subroutine will use when reading a DATA field. The normal value for the RWTS is **0**. The value to change the checksum to should be the next DATA element.

Name: **THE SECTOR EDITOR**
Line Number(s): **310 — 340**
Entry Conditions: Proper DATA pointers and Elements
T1 = lowest track in buffer
TK = highest track in buffer

Function: Automatically performs sector edits as the copy process goes on. It must be called (via GOSUB) just after reading a range of tracks. To indicate how many sector edits are to be performed, you must have a DATA element that has the number of sector edits followed by the word **CHANGES**. For example:

```
1100 DATA 7 CHANGES,1,1,3,4
```

would tell the sector editor that the next 28 DATA elements are sector edits. This is because each sector edit is defined in four DATA elements. The location of the 'x CHANGES' element in the DATA string does not matter because the sector editor will search it out and use the elements immediately following it.

The format for the four bytes that define a sector edit is: TRACK, SECTOR, BYTE, CHANGE TO. Each element is decimal and should be within the correct ranges since no error checking is done.

If you use the 'R/W A Range Quickly' routine and you wish to perform some sector edits, you must set **FAST** equal to **1** so that this routine will be able to locate the specified sector in memory.

Name: **EXCHANGE RWTS's**
Line Number(s): **36Ø**
Entry Conditions: A RWTS at $19ØØ

Function: This is the standard swap RWTS's routine. It uses a routine in IOB.OBJØ to exchange the RWTS at $19ØØ with that which is located at $B8ØØ, the normal location for an RWTS. To tell the machine language swap routine (invoked by a CALL 832) what to exchange, a few POKE's must be executed. They are:
POKE 253, *start of first location*
POKE 255, *start of second location*
POKE 224, *number of pages to exchange* (a standard RWTS is eight pages long)

Name: **FORMAT DISK**
Line Number(s): **38Ø — 41Ø**
Entry Conditions:
S2= slot of disk to format
D2= drive number of disk to format

Function: Formats the target disk. It was meant to be used before the controller takes hold of Super IOB (and is GOSUBed by 'Configuration Time') but can be called by the controller should the need arise.

Name: **PRINT TRACK & SECTOR #**
Line Number(s): **43Ø**
Entry Conditions:
TK= The track number to display
ST= The sector number to display

Function: Puts the current track and sector number at the top of the screen in hexadecimal during the softkey operation. It should be invoked just before reading or writing each sector.

Name: **CENTER MESSAGE**
Line Number(s): **45Ø**
Entry Conditions: **A$**= The message

Function: Prints a message in the center of the screen at the current VTAB position. Care should be taken that the message to print is not longer than 40 characters. If so, an error will occur.

Name: **PRINT MESSAGE AND WAIT**
Line Number(s): **470**
Entry Conditions:
A$= The message

Function: Uses 'Center Message' to print the intended message at a **VTAB 11** where it prints *PRESS ANY KEY TO CONTINUE.* It waits for a keypress before RETURNing.

Name: **TOGGLE READ/WRITE**
Line Number(s): **490 — 530**
Entry Conditions:
CD= current command code

Function: Toggles the state of **CD** (from ReaD to WRite and vice versa) and prints the current mode in flashing letters at the very top of the screen. In addition, if the user has only one drive, it asks him to swap disks. It then exits via 'INITIAL IOB SETUP,' thus making the sector buffer ready for the next operation.

Name: **IGNORE UNREADABLE SECTORS**
Line Number(s): **550 — 590**
Entry Conditions: Not Applicable

Function: If the controller should pay no attention to unreadable sectors, then somewhere in the beginning of it should be an **ONERR GOTO 550**. This is used usually with RWTS.13 (since *DOS 3.2* sectors are unreadable until they have been written to) but can be used with any disk that has unreadable sectors which should be ignored.

Note that this routine will ***not*** *function correctly if you are using 'R/W A Range Quickly'. To ignore errors when using this routine, insert a* **POKE 775,96** *into the beginning of your controller.*

The Remainder Of The Program

Lines 1000—9999 are meant for the controller and all DATA statements it contains. All lines greater than 9999 are used by the error trapper or the configurer which consists of all the prompts when the program is run.

The error-trapper will print a disk error and stop the program. If the error wasn't a disk error, the error-trapper will let it occur.

Memory Usage

Before actually looking at some controllers, let's say a few words about memory usage.

Following is a memory allocation table for the various parts of Super IOB. It is extremely important to stay within the boundaries when writing a controller. If not, horrible things may happen (the least of which would be the production of an incorrect copy).

Memory Allocation Table

$0800.$18FF	(2048 — 6399)	intended for the Applesoft part of Super IOB
$1900.$20FF	(6400 — 8447)	space allocated for a moved RWTS.
$2100.$26FF	(8448 — 9983)	Super IOB Applesoft variable space
$2700.$96FF	(9984 — 38655)	enough space for 7 tracks, this is the sector buffer

First, notice the amount of space available for the BASIC program. In view of the space requirement, the end of program should be checked by typing:

```
PRINT PEEK(175) + PEEK(176) * 256
```

Before a controller with an alternate RWTS (Swap controller, etc.) is used. If it has exceeded the 6399 limit, I suggest DELeting all subroutines not referenced by the controller and all REM lines until it fits within the allocated space.

Second, observe the 1534 bytes for variables. This should be enough space for the simple softkey procedure. It is impossible to allocate more memory for variables and use an alternate RWTS file. If you find that you need more memory and the program does not use RWTS.13 or some other moved RWTS, the **LOMEM: 8448** command may be removed from Line **60**. This will allocate what isn't used (by the BASIC program) of the 2K area reserved for the relocated RWTS as variable space.

Never omit the 'HIMEM:' statement!

This could cause variables to overflow into the sector buffer, thus making a faulty copy.

With all this new knowledge, we are finally ready to scrutinize some sample controller programs. Keep in mind that protection schemes can be used with one another. Therefore, a more sophisticated controller for Super IOB will probably be required for most softkeys. Even so, developing new controllers isn't difficult.

The New Standard

If you were to install any of the previous controllers printed by Hardcore COMPUTIST into the new Super IOB, they would work fine, but would be considerably slower than a controller designed for v1.5. To take advantage of the new subroutines in Super IOB v1.5, a new standard controller (the building block controller that only copies normal disks) is necessary. This new controller has been named **The Fast Controller**. The Fast Controller, as well as a description of its operation, follows.

Fast Controller

```
1ØØØ REM FAST CONTROLLER
1Ø1Ø TK = Ø :LT = 35 :ST = 15 :LS = 15 :CD = WR :FAST = 1
1Ø2Ø GOSUB 49Ø : GOSUB 61Ø
1Ø3Ø GOSUB 49Ø : GOSUB 61Ø : IF PEEK (TRK ) = LT THEN 1Ø5Ø
1Ø4Ø TK = PEEK (TRK ) :ST = PEEK (SCT ) : GOTO 1Ø2Ø
1Ø5Ø HOME : PRINT "COPYDONE" : END
```

Line Explanation

1ØØØ *identifies the controller:*

1Ø1Ø *initializes variables:*

TK = Ø, ST = 15 sets the starting sector to be copied at Track Ø, Sector 15.

LT = 35, LS = 15 sets the last sector to be copied at Track 34, Sector Ø.

CD = WR sets the command code to write.

FAST = 1 tells the sector editor subroutine that the sectors are stored in memory in a decreasing order.

1Ø2Ø *the read routine:* toggles read/write to read, Then reads a chunk of sectors

1Ø3Ø *The write routine:* toggles read/write to write, then writes a chunk of sectors. If the last track was written then go to exit routine.

1Ø4Ø *update track and sector number* for next read and go to read routine

1Ø5Ø *The exit routine* clears the screen, prints ending message and exits to Applesoft

How About a NewSwap?

By adding a couple of GOSUB's to Fast Controller you can derive The NewSwap Controller which is a controller that reads with one RWTS and writes with the normal RWTS. This controller is very handy and is used frequently by articles which appear in Hardcore COMPUTIST.

NewSwap Controller

```
1ØØØ REM NEW SWAP CONTROLLER
1Ø1Ø TK = Ø :LT = 35 :ST = 15 :LS = 15 :CD = WR :FAST = 1
1Ø2Ø GOSUB 36Ø : GOSUB 49Ø : GOSUB 61Ø
1Ø3Ø GOSUB 36Ø : GOSUB 49Ø : GOSUB 61Ø : IF PEEK (TRK ) = LT THEN 1Ø5Ø
1Ø4Ø TK = PEEK (TRK ) :ST = PEEK (SCT ) : GOTO 1Ø2Ø
1Ø5Ø HOME : PRINT "COPYDONE" : END
1ØØ1Ø PRINT CHR$ (4 ) "BLOAD RWTS.insert name here,A$19ØØ"
```

Note that the *filename* of the BLOAD command in line 1ØØ1Ø will need to be changed to the name you've given the RWTS of the protected disk.

Saving the Controller

It is recommended that you have easily accessible copies of both of these controllers because controllers printed for other disks will be very similar to these controllers.

Closing Notes

Every controller printed by Hardcore COMPUTIST so far, will work with Super IOB v1.5. However, controllers printed in the future may not work with older versions of Super IOB.

Now, go out there and break some disks!

IOB Variables List

A........... General temporary usage, scrambled by 'Move S Phases' and 'The Sector Editor.'

A$......... holds message to pass to the user via 'Center Message' and 'Print Message And Wait' and is scrambled by 'Toggle Read/Write.'

A1,A2,A3,A4. scrambled by 'Altered Address Marks', 'Altered Data Marks', 'The Sector Editor', 'Altered Ending Marks' and 'Altered Data Checksum' these are READ from DATA statements and POKEd into the appropriate RWTS to change it.

B$......... altered only by 'Configuration Time.'

BF......... Buffer Full holds the status of the sector buffer and is set to 1 if the buffer is either full or empty and 0 if neither. Is changed only by 'R/W SECTOR.'

BUF....... BUFfer constant holds the address where the RWTS is expecting to find the page number of the sector and is used by 'Initial IOB Setup' and 'R/W Sector'. A **PEEK (BUF)** will return the current sector buffer page number.

CD........ CommanD code is used by the controller, 'Toggle Read/Write' and 'R/W Sector' and holds the current RWTS command code (*see RD, WR, and INIT*).

CMD...... CoMmanD code constant holds the address where the RWTS is expecting to find the previously stated command code and is used by 'R/W Sector'. A **POKE CMD,CD** will change the IOB command.

D1......... Drive 1, set during configuration to the drive number of the source drive, is used by 'Toggle Read/Write.'

D2......... Drive 2, same as above except for target drive.

DOS....... Disk Operating System specifies the number of sectors to read or write and is initialized to 16.

DRV...... DRiVe constant holds the address where the RWTS is expecting to find the drive number of the drive to be accessed and is used by 'Initial IOB Setup'. A **PEEK (DRV)** will return the drive last accessed.

DV .... current DriVe, used by 'Initial IOB Setup', 'Toggle Read/Write', and 'Move S Phases', holds the drive number of the drive to be accessed next.

ERR ...... **ERR**or code is used by 'Disk Error' to determine the error that has just occurred.

FAST ..... used by 'The Sector Editor' to calculate the correct addresses of specified sectors if set to **0**. The 'The Sector Editor' then assumes that all sectors are stored in consecutive memory pages in an ascending order. If set to a **1**, 'The Sector Editor' routine assumes that the sectors are stored in a decreasing order (*see the section describing the operation of 'R/W A Range of Sectors'*).

INIT ...... **INIT**ialize command code. A **CD = INIT** will set the command code to format the diskette.

IO .... Input/Output constant holds a **768** (set during configuration) and is **CALL**ed by 'R/W Sector' to induce the RWTS subroutine.

LS .... Last Sector number is used to tell 'R/W A Range Quickly' what sector is the last sector to be read.

LT .... Last Track number is used to tell 'R/W A Range Quickly' the last track to be read or written.

MB .... Maximum Buffer page holds the last page of memory for the sector buffer, is used by 'R/W Sector', is initialized (during configuration) to **151** and should be changed to **130** only when a 13-sector disk is read or written.

OVL ...... Old VoLume constant. A **PEEK (OVL)** will return the volume number of the previously accessed diskette.

PH .... current **PH**ase. If 'MOVE S PHASES' is referenced (by the controller), this variable must contain the disk arms current phase number **(PH = 2 * TK)**.

RD .... **ReaD** command code. A **CD = RD** will set the command to read the disk.

S .... Step is used to tell 'Move S Phases' how many phases to step through **(-120 to 120)**.

S1 .... Slot **1** sets the slot number of the source drive during configuration and is used by 'Toggle Read/Write'.

S2.......... Slot 2. Same as above except for target drive.

SCT....... SeCTor number constant holds the address where the RWTS is expecting to find the sector to be accessed and is used by 'R/W Sector' to tell the RWTS which sector is to be read or written. A **PEEK (SCT)** will return the last accessed sector number.

SLT....... SLoT number constant holds the address where the RWTS is expecting to find the slot number of the disk to be accessed next and is used by 'Initial IOB Setup'. A **PEEK (SLT)** will return the last accessed disk slot number.

SO......... SlOt number is used by 'Toggle Read/Write' and 'Initial IOB Setup' and holds the slot number of the disk to be accessed next.

ST......... SecTor number is used by the controller to tell 'R/W Sector' which sector number is to be read or written next and is also used to tell 'R/W A Range Quickly' the starting sector to be read or written.

TK........ TracK number is used by the controller to tell 'R/W Sector' which track is to be accessed next and is also used to tell 'R/W A Range Quickly' the starting track to read or write.

TRK...... TRacK number constant holds the memory location where the RWTS is expecting to find the track to be accessed. A **PEEK (TRK)** will return the last accessed track number.

VL......... VoLume number is used by the controller to tell 'Toggle Read/Write' (which passes it to 'Initial IOB Setup') the volume number of the disk to be accessed next.

VL$....... altered only by 'Format Disk'.

VOL...... VOLume number constant holds the memory location where the RWTS is expecting to find the volume to be accessed. A **PEEK (VOL)** will return the volume number last used by the controller.

WR........ WRite command code. A **CD** = **WR** will set the command to write.

Super IOB Applesoft BASIC Listing

```
10 REM *********************************************************
20 REM ***                   SUPER IOB 1.5                   ***
30 REM ***                   BY RAY DARRAH                   ***
40 REM *********************************************************
50 REM SET HIMEM BELOW BUFFER AND SET LOMEM ABOVE THE BLOADED RWTS
60 LOMEM: 8448 : HIMEM: 9983 : GOTO 10010
70 REM INITIAL IOB SETUP
80 POKE BUF, 39 : POKE DRV, DV : POKE VOL, VL : POKE SLT, SO * 16 : RETURN
90 REM R/W SECTOR
100 BF = 0 : POKE TRK, TK : POKE SCT, ST : POKE CMD, CD : CALL IO : POKE
    BUF, PEEK (BUF ) + 1 : IF PEEK (BUF ) = > MB THEN BF = 1
110 RETURN
120 REM MOVE S PHASES
130 POKE 49289 + SO * 16 + DV, 0 : POKE 49289 + SO * 16, 0 :A = PH -
    INT (PH / 4 ) * 4 : POKE 1144, 128 + A : POKE 811, 128 + S + A :
    POKE 813, SO * 16 : CALL 810 : POKE 49288 + SO * 16, 0 :PH = PH
    + S : IF PH < 0 THEN PH = 0
140 RETURN
150 REM 16 SECTOR RWTS ALTERATIONS
160 REM ALTERED ENDING MARKS
170 READ A1, A2, A3, A4 : POKE 47505, A1 : POKE 47515, A2 : POKE 47413,
    A3 : POKE 47423, A4 : RETURN
180 REM ALTERED ADDRESS MARKS
190 READ A1, A2, A3 : POKE 47445, A1 : POKE 47455, A2 : POKE 47466, A3
    : RETURN
200 REM ALTERED DATA MARKS
210 READ A1, A2, A3 : POKE 47335, A1 : POKE 47345, A2 : POKE 47356, A3
    : RETURN
220 REM NORMALIZER
230 POKE 47505, 222 : POKE 47515, 170 : POKE 47413, 222 : POKE 47423, 170
240 POKE 47445, 213 : POKE 47455, 170 : POKE 47466, 150 : POKE 47335, 213
250 POKE 47345, 170 : POKE 47356, 173 : POKE 47360, 0 : POKE 47498, 183
    : RETURN
260 REM IGNORE ADDRESS CHECKSUM
270 POKE 47498, 0 : RETURN
280 REM ALTERED DATA CHECKSUM
290 READ A1 : POKE 47360, A1 : RETURN
300 REM THE SECTOR EDITOR
310 READ A$ : IF RIGHT$ (A$, 7 ) < > "CHANGES" THEN 310
320 FOR A = 1 TO VAL (A$ ) : READ A1, A2, A3, A4
330 IF A1 < T1 OR A1 > TK THEN NEXT : RETURN
340 POKE 9984 + (A1 - T1 ) * 4096 + ABS (FAST * 15 - A2 ) * 256 + A3,
    A4 : NEXT : RETURN
350 REM SWAP RWTS AT $1900 WITH THE ONE AT $B800
360 POKE 253, 25 : POKE 255, 184 : POKE 224, 8 : CALL 832 : RETURN
370 REM FORMAT DISK
```

```
380 A$ = "VOLUME^NUMBER^FOR^COPY^=>254" : HOME : GOSUB 45Ø : HTAB 32
     : INPUT "" ;VL$ :VL = VAL (VL$ ) : IF VL$ = "" THEN VL = 254
39Ø IF VL > 255 OR VL < Ø THEN 38Ø
4ØØ POKE CMD, INIT :SO = S2 :DV = D2 :A$ = "INSERT^BLANK^DISK^IN^
     SLOT^" + STR$ (S2 ) + ",^DRIVE^" + STR$ (D2 ) : GOSUB 47Ø
41Ø GOSUB 8Ø : HOME :A$ = "FORMATTING" : FLASH : GOSUB 45Ø : NORMAL :
     CALL IO :VL = Ø : RETURN
42Ø REM PRINT TRACK & SECTOR
43Ø VTAB 3 : HTAB 1Ø : PRINT "TRACK^$" MID$ (HX$, TK * 2 + 1, 2 ) "^^
     SECTOR^$" MID$ (HX$, ST * 2 + 1, 2 ) "^^" : RETURN
44Ø REM CENTER MESSAGE
45Ø HTAB 21 - LEN (A$ ) / 2 : PRINT A$; : RETURN
46Ø REM PRINT MESSAGE AND WAIT
47Ø HOME : VTAB 11 : GOSUB 45Ø : VTAB 13 :A$ = "PRESS^ANY^KEY^TO^
     CONTINUE" : GOSUB 45Ø : WAIT - 16384, 128 : GET A$ : RETURN
48Ø REM TOGGLE READ/WRITE
49Ø CD = (CD = 1 ) + 1 : IF CD = RD THEN A$ = "INSERT^SOURCE^DISK." :SO
     = S1 :DV = D1 : GOTO 51Ø
5ØØ A$ = "INSERT^TARGET^DISK." :SO = S2 :DV = D2
51Ø IF D1 = D2 AND S1 = S2 THEN GOSUB 47Ø : HOME
52Ø VTAB 1 : HTAB 1 : PRINT SPC( 39 ); : FLASH :A$ = "READING" : IF CD =
     WR THEN A$ = "WRITING"
53Ø GOSUB 45Ø : NORMAL : GOTO 8Ø
54Ø REM ONERR IGNORE UNREADABLE SECTORS
55Ø CALL 822 :ERR = PEEK (222 )
56Ø IF ERR = 255 OR ERR = 254 OR CD < > RD THEN 1Ø23Ø
57Ø IF ERR > 15 THEN POKE 216, Ø : RESUME
58Ø PRINT CHR$ (7 ); : POKE BUF, PEEK (BUF ) + 1 : IF PEEK (BUF ) = > MB
     THEN BF = 1
59Ø RETURN
6ØØ REM R/W A RANGE QUICKLY
61Ø PR# Ø : IN# Ø : POKE 86Ø, MB : POKE 861, LT : POKE 862, LS
62Ø POKE CMD, CD : POKE TRK, TK : POKE SCT, ST : GOSUB 43Ø : CALL 863 :
     CALL 1ØØ2 : RETURN
1ØØØØ REM CONFIGURATION TIME
1ØØ1Ø REM BLOAD RWTS HERE
1ØØ2Ø IF PEEK (768 ) * PEEK (769 ) = 5Ø7 THEN 1ØØ6Ø
1ØØ3Ø HOME :A$ = " *^SUPER^IOB^*": GOSUB 45Ø : PRINT : PRINT :A$ =
     "CREATED^BY^RAY^DARRAH" : GOSUB 45Ø
1ØØ4Ø VTAB 1Ø :A$ = "INSERT^SUPER^IOB^DISK" : GOSUB 45Ø :T : PRINT :
     PRINT :A$ = "PRESS^ANY^KEY^TO^CONTINUE" : GOSUB 45Ø : WAIT -
     16384, 128 : GET A$
1ØØ5Ø PRINT : PRINT CHR$ (4 ) "BLOAD^IOB.OBJØ,A$3ØØ"
1ØØ6Ø TK = ST = VL = CD = DV = SO :RD = 1 :WR = 2 : INIT = 4 : ONERR GOTO 1Ø22Ø
1ØØ7Ø IO = 768 :SLT = 779 :DRV = 78Ø :VOL = 781 :TRK = 782 :SCT = 783 :BUF
     = 787 :CMD = 79Ø :OVL = 792
1ØØ8Ø HOME :DOS = 16 :MB = 151 :HX$ = "ØØØ1Ø2Ø3Ø4Ø5Ø6Ø7Ø8Ø9ØAØBØ
     CØDØEØF1Ø1112131415161718191A1B1C1D1E1F2Ø2122"
1ØØ9Ø VTAB 8 : PRINT :A$ = "ORIGINAL" :S2 = 6 :D2 = 1 : GOSUB 1Ø14Ø :S1
     = S2 :D1 = D2
```

```
1Ø1ØØ PRINT : PRINT : PRINT :D2 = (D2 = 1 ) + 1 :A$ = "DUPLICATE" : GOSUB
    1Ø14Ø
1Ø11Ø A$ = "FORMAT^BACK^UP^FIRST?^N" + CHR$ (8 ) : HOME : VTAB 12 :
    GOSUB 45Ø : GET A$ : IF A$ = "Y" THEN GOSUB 38Ø
1Ø12Ø HOME :A$ = "INSERT^DISKS^IN^PROPER^DRIVES." : GOSUB 47Ø : HOME
    : GOTO 1ØØØ
1Ø13Ø REM GET SLOT AND DRIVE
1Ø14Ø GOSUB 45Ø : PRINT : PRINT : PRINT TAB( 1Ø ) "SLOT=>" S2 SPC( 8 )
    "DRIVE=>" D2;
1Ø15Ø HTAB 16 :B$ = "7" : GOSUB 1Ø18Ø :S2 = VAL (A$ )
1Ø16Ø HTAB 32 :B$ = "2" : GOSUB 1Ø18Ø :D2 = VAL (A$ ) : RETURN
1Ø17Ø REM GET A KEY
1Ø18Ø GET A$ : IF (A$ < "1" OR A$ > B$ ) AND A$ < > CHR$ (13 ) THEN 1Ø18Ø
1Ø19Ø IF A$ = CHR$ (13 ) THEN A$ = CHR$ ( PEEK ( PEEK (4Ø ) + PEEK (41 )
    * 256 + PEEK (36 ) ) - 128 )
1Ø2ØØ PRINT A$; : RETURN
1Ø21Ø REM DISK ERROR
1Ø22Ø ERR = PEEK (222 ) : IF ERR > 15 AND ERR < 254 THEN POKE 216 , Ø : CALL
    822 : RESUME
1Ø23Ø IF ERR = 254 THEN PRINT "TYPE^AGAIN^PLEASE:" : PRINT : RESUME
1Ø24Ø IF ERR = 255 THEN STOP
1Ø25Ø IF ERR = Ø THEN A$ = "INITIALIZATION^ERROR"
1Ø26Ø IF ERR = 1 THEN A$ = "WRITE^PROTECTED"
1Ø27Ø IF ERR = 2 THEN A$ = "VOLUME^MISMATCH^ERROR"
1Ø28Ø IF ERR = 4 THEN A$ = "DRIVE^ERROR"
1Ø29Ø IF ERR = 8 THEN A$ = "READ^ERROR"
1Ø3ØØ VTAB 12 : GOSUB 45Ø : PRINT CHR$ (7 ) : END
```

Super IOB v1.5 Hex Dump

```
Ø3ØØ: A9 Ø3 AØ ØA 2Ø D9 Ø3 BØ
Ø3Ø8: 16 6Ø Ø1 6Ø Ø1 ØØ ØØ ØØ
Ø31Ø: 1B Ø3 ØØ 27 ØØ ØØ ØØ ØØ
Ø318: ØØ 6Ø Ø1 ØØ Ø1 EF D8 AD
Ø32Ø: 17 Ø3 4A 4A 4A 4A AA 4C
Ø328: 12 D4 A9 ØØ A2 ØØ 4C AØ
Ø33Ø: B9 A6 DE 4C 12 D4 68 A8
Ø338: 68 A6 DF 9A 48 98 48 6Ø
Ø34Ø: AØ ØØ 84 FC 84 FE B1 FC
Ø348: 48 B1 FE 91 FC 68 91 FE
Ø35Ø: C8 DØ F3 E6 FD E6 FF C6
Ø358: EØ DØ EB 6Ø 97 35 ØF A9
Ø36Ø: ØØ 85 28 A9 Ø5 85 29 A9
Ø368: 1Ø 85 24 AD ØE Ø3 2Ø DA
Ø37Ø: FD A9 1C 85 24 AD ØF Ø3
Ø378: 2Ø DA FD 2Ø ØØ Ø3 CE Ø3
Ø38Ø: Ø3 1Ø Ø8 A9 ØF 8D ØF Ø3
Ø388: EE ØE Ø3 EE 13 Ø3 AD 13
Ø39Ø: Ø3 CD 5C Ø3 BØ C5 AD ØF
Ø398: Ø3 CD 5E Ø3 DØ C9 AD ØE
Ø3AØ: Ø3 CD 5D Ø3 9Ø C1 BØ B3
```

Super IOB v1.5 Source Code

Super IOB machine routines
BY RAY DARRAH

```
Ø3D9- RWTS.B8ØØ  .EQ $Ø3D9    ENTRY POINT TO RWTS @$B8ØØ
D412- INVOKERROR .EQ $D412    ROUTINE THAT CAUSES BASIC TO DO THE ERROR
                              CONTAINED IN X
1EØØ- RWTS.19ØØ  .EQ $1EØØ    ENTRY POINT TO THE RWTS AT $19ØØ
B9AØ- SEEKABS    .EQ $B9AØ    ENTRY POINT TO THE SEEKABS ROUTINE AT $B8ØØ
ØØDE- BAS.ERR    .EQ 222      BASIC ON ERR ERROR CODE
ØØFC- SWFRM      .EQ $FC      EXCHANGE FROM PARAMTER
ØØFE- SWTO       .EQ $FE      EXCHANGE RWTS 'TO' PARAMETER
ØØEØ- PAGES      .EQ $EØ      NUMBER OF PAGES OF MEMORY TO EXCHANGE
ØØ24- CH         .EQ $24      CURSOR X POSITION
FDDA- PRNTBYTE   .EQ $FDDA    PRINTS HEXADECIMAL BYTE
                 .OR $Ø3ØØ    STARTS AT PAGE THREE
 *               .TF          IOB.OBJØ                                 *
```

CALL RWTS

```
Ø3ØØ:  A9 Ø3     IO         LDA /TABLETYP   ENTRY POINT FOR CALLING THE RWTS
                                            THROUGH BASIC
Ø3Ø2:  AØ ØA                LDY #TABLETYP   A,Y POINT TO THE IOB TABLE
Ø3Ø4:  2Ø D9 Ø3             JSR RWTS.B8ØØ   GO TO THE RWTS AT $B8ØØ
Ø3Ø7:  BØ 16                BCS DOS.ERR     CAUSE BASIC ERROR IF CARRY SET
Ø3Ø9:  6Ø                   RTS             OTHERWISE, ALL IS WELL SO RETURN
Ø3ØA:  Ø1        TABLETYP   .HS Ø1          TYPE OF TABLE (1=IOB)
Ø3ØB:  6Ø        SLT        .HS 6Ø          NEXT ACCESSED SLOT (VIA POKESLT,SO)
Ø3ØC:  Ø1        DRV        .HS Ø1          DRIVE TO BE ACCESSED NEXT (1 OR 2)
Ø3ØD:  ØØ        VOL        .HS ØØ          VLUME TO BE ACCESSED (Ø=ANYTHING)
Ø3ØE:  ØØ        TRK        .HS ØØ          TRACK TO ACCESS
Ø3ØF:  ØØ        SCT        .HS ØØ          SECTOR TO ACCESS
Ø31Ø:  1B Ø3     DCTPTR     .DA DCT         DEVICE CHARACTERISTICS TABLE POINTR
Ø312:  ØØ        BUFFERLO   .HS ØØ          ALWAYS MAKE LSB OF BUFFER POINTER Ø!
Ø313:  27        BUF        .HS 27          SECTOR BUFFER PAGE POINTER
Ø314:  ØØ        NOTHING    .HS ØØ          NOT USED
Ø315:  ØØ        BYTCOUNT   .HS ØØ          PARTIAL SECTOR BYTE COUNT (Ø=256
                                            BYTES)
Ø316:  ØØ        CMD        .HS ØØ          COMMAND CODE (Ø=SEEK)
Ø317:  ØØ        RWTS.ERR   .HS ØØ          ERROR CODE THAT RWTS.B8ØØ RETURNED
Ø318:  ØØ        OVL        .HS ØØ          VOLUME NO. OF LAST ACCESSED DISK
Ø319:  6Ø        OLDSLT     .HS 6Ø          SLOT PREVIOUSLY ACCESSED
Ø31A:  Ø1        OLDDRV     .HS Ø1          DRIVE PREVIOUSLY ACCESSED
Ø31B:  ØØ        DCT        .HS ØØ          TYPE OF DEVICE CHARACTERISTICS
                                            TABLE
Ø31C:  Ø1        PHASES     .HS Ø1          PHASES-1 PER TRACK, (Ø OR 1)
Ø31D:  EF D8     MOTORCNT   .HS EFD8        MOTOR-ON TIME COUNT
Ø31F:  AD 17 Ø3  DOS.ERR    LDA RWTS.ERR    DOS HAD AN ERROR, GET THE ERROR CODE
Ø322:  4A                   LSR             DIVIDE IT BY 16
```

```
Ø323:  4A                   LSR
Ø324:  4A                   LSR
Ø325:  4A                   LSR
Ø326:  AA                   TAX              TRANSFER IT TD X SO BASIC WLL INDUCE
                                             THE FALSE ERROR CODE
Ø327:  4C 12 D4             JMP INVDKERRDR   CAUSE A BASIC ERRDR
```

MOVE THE DISK ARM

```
Ø32A:  A9 ØØ    MDVPHASES   LDA #$ØØ         ROUTINE TD SET UP REGISTERS BEFDRE
                                             CALLING SEEKABS
Ø32C:  A2 ØØ                LDX #$ØØ         X AND A HAVE DUMMY NUMBERS THAT WILL
                                             BE PDKED INTD BY "MOVE S PHASES"
Ø32E:  4C AØ B9             JMP SEEKABS
```

CAUSE ERROR IN CONTROLLER

```
Ø331:  A6 DE    BASICERR    LDX BAS.ERR      BASIC HAS MADE AN ERRDR SD CAUSE THE
                                             ERROR NUMBER AT 222
Ø333:  4C 12 D4             JMP INVDKERROR
```

POP OFF RETURN

```
Ø336:  68       PDP         PLA              ROUTINE TD POP DFF DNE RETURN
                                             (BASIC) ADDRESS
Ø337:  A8                   TAY
Ø338:  68                   PLA
Ø339:  A6 DF                LDX BAS.ERR+1    GET WHAT STACK WOULD BE IF GDSUB
                                             WASN'T THERE
Ø33B:  9A                   TXS              PUT THAT AS THE STACK POINTER
Ø33C:  48                   PHA
Ø33D:  98                   TYA              RESTDRE THE LAST RETURN ADDRESS
Ø33E:  48                   PHA
Ø33F:  6Ø                   RTS
```

EXCHANGE RWTS's

```
Ø34Ø:  AØ ØØ                LDY #Ø           ZERD THE LSB's
Ø342:  84 FC                STY SWFRM        AND HAVE Y AT ZERO FDR START
Ø344:  84 FE                STY SWTD
Ø346:  B1 FC    MOVE.PAGE   LDA (SWFRM),Y    GET A BYTE
Ø348:  48                   PHA              AND SAVE IT
Ø349:  B1 FE                LDA (SWTD),Y     GET THE BYTE WHERE THE SAVED ONE GOES
Ø34B:  91 FC                STA (SWFRM),Y    AND STDRE IT WHERE THE SAVED ONE WAS
Ø34D:  68                   PLA              GET THE SAVED BYTE
Ø34E:  91 FE                STA (SWTD),Y     AND STORE IT WHERE IT GDES
Ø35Ø:  C8                   INY              DONE WITH A PAGE
Ø351:  DØ F3                BNE MOVE.PAGE    NO KEEP WORKING ON IT
```

```
Ø353:  E6 FD                INC SWFRM+1     GET NEXT MSB's
Ø355:  E6 FF                INC SWTD+1
Ø357:  C6 EØ                DEC PAGES       DECREMENT THE NUMBER DF PAGES TO
                                            MOVE
Ø359:  DØ EB                BNE MOVE.PAGE   IF NOT DONE, MOVE ANOTHER PAGE
Ø35B:  6Ø         RTS1      RTS             FINISHED, RTS
```

READ OR WRITE THE ENTIRE BUFFER

```
Ø35C:  97         MB        .HS 97          HIGHEST BUFFER PAGE+1
Ø35D:  35         LT        .HS 35          LAST TRACK TO GET+1
Ø35E:  ØF         LS        .HS ØF          (LAST SECTOR TO GET-1)MOD16
Ø35F:  A9 ØØ                LDA #Ø          DO A VTAB3
Ø361:  B5 2B                STA $28
Ø363:  A9 Ø5                LDA #5
Ø365:  B5 29                STA $29
Ø367:  A9 1Ø      NEWTRK    LDA #16         HTAB
Ø369:  B5 24                STA CH
Ø36B:  AD ØE Ø3             LDA TRK
Ø36E:  2Ø DA FD             JSR PRNTBYTE    PRINT IT
Ø371:  A9 1C                LDA #2B         HTAB
Ø373:  B5 24                STA CH
Ø375:  AD ØF Ø3             LDA SCT
Ø37B:  2Ø DA FD             JSR PRNTBYTE
Ø37B:  2Ø ØØ Ø3             JSR IO          GET A SECTOR
Ø37E:  CE ØF Ø3             DEC SCT         NEXT SECTOR
Ø381:  1Ø Ø8                BPL NXT.PG
Ø3B3:  A9 ØF                LDA #15         RESTORE TD SECTDR 15
Ø3B5:  BD ØF Ø3             STA SCT
Ø3BB:  EE ØE Ø3             INC TRK         NEXT TRK
Ø3BB:  EE 13 Ø3   NXT.PG    INC BUF         NEXT PAGE OF MEMDRY
Ø3BE:  AD 13 Ø3             LDA BUF
Ø391:  CD 5C Ø3             CMP MB
Ø394:  BØ C5                BCS RTS1        BUFFER FULL, RETURN
Ø396:  AD ØF Ø3             LDA SCT         LAST SECTOR?
Ø399:  CD 5E Ø3             CMP LS
Ø39C:  DØ C9                BNE NEWTRK
Ø39E:  AD ØE Ø3             LDA TRK
Ø3A1:  CD 5D Ø3             CMP LT
Ø3A4:  9Ø C1                BCC NEWTRK
Ø3A6:  BØ B3                BCS RTS1        END DF DISK
```

Using ProDOS on a Franklin Ace

Hardcore COMPUTIST # 9, page 18

The March 1984 issue of *Assembly Lines* showed how to boot ProDOS on a Franklin Ace computer by NOPping two bytes in the ProDOS system file after it had been loaded into memory. However, this method would not work for the ProDOS file dated 1-JAN-84.

The ProDOS system file contains a checksum-like subroutine which returns a value of $ØC if a genuine Apple is detected and a $ØØ otherwise. If a non-Apple is detected, ProDOS will just hang and not load in the BASIC.SYSTEM interpreter. A disassembly of this routine from this ProDOS file looks like:

```
2639-  18          CLC           2652-  ØA          ASL
263A-  AC  31  26   LDY $2631     2653-  ØA          ASL
263D-  B1  ØA       LDA ($ØA),Y   2654-  ØA          ASL
263F-  29  DF       AND #DF       2655-  A8          TAY
2641-  6D  31  26   ADC $2631     2656-  4D  31  26   EOR $2631
2644-  8D  31  26   STA $2631     2659-  69  ØB       ADC #$ØB
2647-  2E  31  26   ROL $2631     265B-  DØ  Ø3       BNE $266Ø
264A-  C8           INY           265D-  A5  ØC       LDA #$ØC
264B-  CC  34  26   CPY $2634     265F-  6Ø          RTS
264E-  DØ  ED       BNE $263D     266Ø-  A9  ØØ       LDA #$ØØ
265Ø-  98           TYA           2662-  6Ø          RTS
2651-  ØA           ASL
```

In order for this routine to always return with a value of $ØØ, the branch to the code which loads the accumulator with $ØØ (LDA #$ØØ) needs to be removed. This can be done by replacing the BNE $26ØØ (DØ Ø3) instruction with two NOPs (EA EA).

The easiest way to do this is to use a sector editor and zap the change directly to the disk. Any sector editor can be used on ProDOS disk because the formatting has not been changed from DOS 3.3. On the ProDOS USERS DISK change bytes $5B and $5C of track $Ø1, sector $Ø9 from DØ Ø3 to EA EA and rewrite the sector.

If you have a sector editor with search capability, such as ZAP, you should search for a byte sequence of 69 ØB DØ Ø3, since the sequence of DØ Ø3 is a fairly common one.

Once you have made this change, ProDOS should boot and operate on Franklin's and other compatibles that have their monitor ROM routines in the proper locations.

Crunchlist II

by Ray Darrah

This is a compilation of two articles by Ray Darrah
Crunchlist *from Hardcore COMPUTIST #6, page 26.*
Crunchlist II *from Hardcore COMPUTIST #10, page 26.*

The machine language program (hex and source code) is for Crunchlist II, the updated version of Crunchlist.

Crunchlist is an 'Ampersand' (**&**) utility designed for maximum disk-space utilization when creating EXECable text files.

An EXECable text file is one in which part (or all) of a program has been saved as keystrokes instead of the usual way, as Applesoft tokens. When done properly, EXECing the file (i.e. typing EXEC *filename*) makes the computer think you are typing in the program lines and so they are inserted between (or replace) the existing lines of the program in memory.

The usual way to capture an EXECable file is to add a line to the program to be captured. This line often looks something like this:

```
1  NM$ = "CAPTURED" :D$ = CHR$( 4) : PRINT D$"OPEN" NM$: PRINT D$"WRITE"
     NM$: POKE 33,33: LIST 1000 - 999: PRINT D$"CLOSE": TEXT: END
```

Notice the **LIST 1000 - 9999** statement. When the Applesoft interpreter comes across this, it does a regular LISTing. As you probably have already noticed, a lot of spaces are inserted into a regular listing to make it more readable. When you capture a program in this manner, all those spaces are saved on the disk.

Since these spaces aren't required when typing line numbers in (even via EXEC), why have them lying around on your disk just taking up space?

Crunchlist eliminates this diskette waste by listing specified line numbers **without unnecessary spaces**! In addition to the disk storage gained because of this, the captured programs take less time to capture and less time to EXEC back in.

Using *Crunchlist*, the above capture line would look like this:

```
1  NM$ = "CAPTURED" :D$ =CHR$( 4) : PRINTD$"OPEN" NM$: PRINTD$"WRITE"
     NM$: & 1000, 999: PRINT D$"CLOSE": END
```

Notice that LIST is replaced by the ampersand (**&**) and a comma (**,**) replaces the dash (–). This is because *Crunchlist* can not only list lines by numbers, it can also list a collection of lines by specifying an expression such as **999 + 1**. The dash will be interpreted as a 'minus' sign! The **POKE 33,33** and **TEXT** statements have been entirely removed because *Crunchlist*, as opposed to **LIST**, doesn't indent lines. Try *Crunchlist*ing some long lines in immediate execution for a good example.

Now you are ready to start capturing EXECable files and conserve disk space at the same time. This is great for saving lines that appear in a lot of your creations (like a title page or REMarks stating who wrote it). I'm sure you can think of a hundred other uses for it also (maybe something with a printer).

Using Crunchlist

First, type in the **Hexdump for CRUNCHLIST II** and:

```
BSAVE CRUNCHLIST,A$300,L$98
```

Whenever you wish to install *Crunchlist*, simply type:

```
BRUN CRUNCHLIST
```

Now, use **&** instead of **LIST** on those lines you wish to capture.

Hexdump for CRUNCHLIST II

```
0300: A9 4C 8D F5 03 A9 10 8D
0308: F6 03 A9 03 8D F7 03 60
0310: 20 A4 03 20 1A D6 A5 9B
0318: 48 A5 9C 48 20 B7 00 F0
0320: 06 20 BE DE 20 A4 03 68
0328: 85 9C 68 85 9B A5 50 05
0330: 51 D0 06 A9 FF 85 50 85
0338: 51 A0 01 B1 9B F0 36 C8
0340: B1 9B AA C8 B1 9B C5 51
0348: D0 04 E4 50 F0 02 B0 25

0350: 84 85 20 24 ED A4 85 4C
0358: 5F 03 09 80 20 ED FD C8
0360: B1 9B D0 12 20 8E FD A0
0368: 00 B1 9B AA C8 B1 9B 86
0370: 9B 85 9C D0 C4 60 10 E2
0378: E9 7F AA 84 85 A0 D0 84
0380: 9D A0 CF 84 9E A0 FF CA
0388: F0 07 20 2C D7 10 FB 30
0390: F6 20 2C D7 30 08 09 80
0398: 20 ED FD 4C 91 03 20 ED

03A0: FD 4C 55 03 20 7B DD 20
03A8: F2 EB A5 A0 85 51 A5 A1
03B0: 85 50 60
```

Source Code for CRUNCHLIST II

```
1ØØØ ***********************************
1Ø1Ø *                                 *
1Ø2Ø *           CRUNCHLIST            *
1Ø3Ø *         BY RAY DARRAH III       *
1Ø4Ø *                                 *
1Ø5Ø ***********************************
115Ø *    APPLESOFT ROUTINES/LOCATIONS *
116Ø
117Ø FINDLIN    .EQ $D61A
118Ø LINPRT     .EQ $ED24    ROUTINE THAT PRINTS X,A IN DECIMAL
119Ø TOKTABL    .EQ $DØDØ    TABLE OF TOKENS SPELLED OUT
12ØØ GETCHR     .EQ $D72C    ROUTINE THAT GETS NEXT CHARACTER OF COMMAND
                             WORD
121Ø FRM.EVAL   .EQ $DD7B    EVALUATES AN EXPRESSION AND PUTS IT IN THE FAC
122Ø INT.CONV   .EQ $EBF2    CONVERTS VALUE IN //C TO INTEGER AND STORES
                             IT IN $AØ.$A1
123Ø CHKCOM     .EQ $DEBE
124Ø
125Ø *                                 *
126Ø *           MONITOR ROUTINES      *
127Ø *                                 *
128Ø
129Ø COUT       .EQ $FDED    PRINT A AS ASCII
13ØØ CROUT      .EQ $FD8E    PRINT A CARRIAGE RETURN
131Ø
132Ø *                                 *
133Ø *         ZERO PAGE LOCATONS      *
134Ø *                                 *
135Ø
136Ø CHRGOT     .EQ $B7      RETRIEVE LAST CHARACTER FROM BASIC
137Ø CHRGET     .EQ $B1      GET A NEW BASIC CHARACTER
138Ø LINNUM     .EQ $5Ø      NUMBER OF LAST LINUMBER TO LIST
139Ø LOWTR      .EQ $9B      POINTER TO CURRENT LINE THAT WE ARE LISTING
14ØØ FORPNT     .EQ $85      TEMPORARY STORAGE OF THE OFFSET FOR THIS LINE
141Ø FAC        .EQ $9D      USED BY GETCHR SHOULD POINT TO WHICH TOKEN
                             YOU ARE GETTING
142Ø
143Ø *                                 *
144Ø *       PAGE THREE LOCATIONS      *
145Ø *                                 *
146Ø
147Ø AMPER      .EQ $3F5     AMPERSAND JMP LOCATION
148Ø
```

```
1490 *                                                        *
1500 *                     START OF PROGRAM                   *
1510 *                                                        *
1520
1530          .OR $300     SQUEEZE IT IN TO PAGE THREE
1540          .TF          CRUNCHLIST.OBJ
1550          LDA #$4C     MAKE AMPERSAND JMP TO START
1560          STA AMPER
1570          LDA #START
1580          STA AMPER+1
1590          LDA /START
1600          STA          AMPER+2
1610          RTS          HOOKUP COMPLETE
1620 START    JSR LINGET   SET LINNUM TO START OF RANGE
1630          JSR FINDLIN  POINT LOWTR TO 1ST LINE
1640          LDA LOWTR    SAVE POINTER
1650          PHA
1660          LDA LOWTR+1
1670          PHA
1680          JSR CHRGOT   RANGE SPECIFIED?
1690          BEQ MAINLST  YES, LIST IT
1700          JSR CHKCOM   SKIP THE COMMA
1710          JSR LINGET   SET LINNUM TO END RANGE
1720 MAINLST  PLA MOVE     LOWTR BACK
1730          STA LOWTR+1
1740          PLA
1750          STA LOWTR
1760          LDA LINNUM   IS THE END NUMBER A NULL?
1770          ORA LINNUM+1
1780          BNE NXLST    IF NO, START LISTING
1790          LDA #$FF     YES IT IS, SO SET THE END NUMBER TO 65535
1800          STA LINNUM
1810          STA LINNUM+1
1820 NXLST    LDY #1       LIST AN ENTIRE LINE
1830          LDA (LOWTR),Y IS HIGH BYTE OF LINK ZERO
1840          BEQ FINISHED YES, DONE LISTING
1850          INY GET      LINE NUMBER
1860          LDA (LOWTR),Y
1870          TAX
1880          INY
1890          LDA (LOWTR),Y
1900          CMP LINNUM+1
1910          BNE LSTD
1920          CPX LINNUM
1930          BEQ LST1LIN
1940 LSTD     BCS FINISHED
1950 LST1LIN  STY FORPNT
1960          JSR          LINPRT
1970 LISTLOOP LDY FORPNT
1980          JMP PRINT1
```

```
1990 SENDCHR     ORA #$80
2000             JSR COUT
2010 PRINT1      INY
2020             LDA (LOWTR),Y
2030             BNE TOKENCMP   NOT END OF LINE SO SEE IF TOKEN
2040             JSR CROUT      EVERY LINE TERMINATED BY A CARRIAGE RETURN
2050             LDY #0         AT END OF LINE TO GET NEXT LINK
2060             LDA (LOWTR),Y
2070             TAX
2080             INY
2090             LDA (LOWTR),Y
2100             STX LOWTR      POINT TO NEXT LINE
2110             STA LOWTR+1
2120             BNE NXLST      BRANCH IF NOT AT END OF PROGRAM
2130 FINISHED    RTS            COMPLETELY DONE LISTING SO RETURN
2140 TOKENCMP    BPL SENDCHR     IF NOT A TOKEN, JUST PRINT IT
2150             SEC
2160             SBC #$7F       MAKE IT A INDEX NUMBER FOR THE TABLE
2170             TAX
2180             STY FORPNT     SAVE LINE POIINTER
2190             LDY #TOKTABL-$100
2200             STY FAC        POINT FAC TO TABLE
2210             LDY /TOKTABL-$100
2220             STY FAC+1
2230             LDY #$FF
2240 SKPTK       DEX            COUNT TOKENS VERSA X
2250             BEQ PRTOK
2260 TOKLP       JSR GETCHR
2270             BPL TOKLP
2280             BMI SKPTK
2290 PRTOK       JSR GETCHR     GET A CHARACTER OF THE TOKEN
2300             BMI TOKDONE
2310             ORA #$80
2320             JSR COUT
2330             JMP PRTOK
2340 TOKDONE     JSR COUT       SEND LAST CHARACTER OF TOKEN
2350             JMP LISTLOOP   ;GO DO NEXT THING IN LINE
2360
2370 LINGET      JSR FRM.EVAL   EVALUATE FORMULA
2380             JSR INT.CONV   CONVERT IT TO INTEGER
2390             LDA $A0        MOVE INTEGER VALUE
2400             STA LINNUM+1
2410             LDA $A1
2420             STA LINNUM
2430             RTS
2440
```

The Controller Saver

by Ray Darrah

Hardcore COMPUTIST # 10, page 11

Just when you thought changing controllers in Super IOB wasn't worth the effort (or the disk space), along comes the **Controller Saver**.

How it works

The *Controller Saver* (named **SAVE CONTROLLER**) extracts the controller portion of Super IOB and saves it to disk as a text file. Once this is done, softkeying a disk is as easy as loading Super IOB and EXECing the controller of your choice.

Saving a controller as a text file (denoted by **T** on the left side of the CATALOG) takes up 1/5 the space that a different Super IOB program for each new controller would. SAVE CONTROLLER extracts Lines 1000—9999 as well as any other lines you may have added or inserted into *Super IOB*.

SAVE CONTROLLER uses CRUNCHLIST (see the **Crunchlist** article). If you haven't typed *CRUNCHLIST* in yet, do so immediately.

Once *CRUNCHLIST* has been BSAVEd, type in the Applesoft listing of MAKE SAVER and:

SAVE MAKE SAVER

on the same disk. Next:

RUN MAKE SAVER

This will put a file on the disk called SAVE CONTROLLER which is an EXECable image of Lines 50000-50220 in the MAKE SAVER program. If you wish to alter the *Controller Saver*, you should make the alterations to these lines and then RUN the program and it will put the image of the new program on the disk.

How Do I Use It?

Hence forth, when you wish to save a controller, simply type

EXEC SAVE CONTROLLER

After some disk spinning and lots of prompts rolling up the screen, the computer will ask:

DELETE ANY LINES?

This is asking if you had to eliminate any line numbers of Super IOB in order for it to fit in the allocated space. If you press [Y], the computer will then ask which lines you DELeted. You must reply with a range of line numbers separated by a comma. If you only DELeted one line then you must type the same number twice (still separated by a comma). When you are finished, type:

0,0

Now, upon the EXECing of this controller, those lines specified will be DELeted from *Super IOB*.

Next you are asked:

SAVE ANY EXTRA LINES?

If your controller occupies more area than Lines 1000—9999, then you must press [Y] and enter the other range(s) (just like the deletions) to be saved (besides the usual 1000 through 9999). Again, you exit this routine by typing:

0,0

The final prompt is:

CONTROLLER FILENAME =>

This is merely a request for the entry of the filename. I usually append a **.CON** to the end of the name to designate it as a controller.

BASIC listing for: MAKE SAVER

```
1Ø D$ = CHR$ (13) + CHR$ (4) :NM$ = "SAVE^CONTROLLER": HOME
2Ø VTAB 12: PRINT TAB(11)"ONE^MOMENT^PLEASE."
3Ø PRINT D$"NOMONCIO"D$"BRUNCRUNCHLIST.OBJ"
4Ø PRINT D$"OPEN"NM$D$"DELETE"NM$D$"OPEN"NM$D$"WRITE"NM$
5Ø PRINT "DEL5ØØØØ,59999": & 5ØØØØ,59999: PRINT "RUN5ØØØØ";
6Ø PRINT D$"CLOSE": HOME : PRINT "CONTROLLER^SAVER^FILE^COMPLETE":
    END
5ØØØØ TEXT : HOME : VTAB 5: PRINT "SUPER^IOB^CONTROLLER^SAVE"
5ØØ1Ø PRINT : PRINT : PRINT CHR$ (4)"BRUNCRUNCHLIST.OBJ": DIM X(1Ø,3)
5ØØ2Ø PRINT "DELETE^ANY^LINES?^N" CHR$ (8);: GET A$: IF A$<>"Y" THEN
    5ØØ5Ø
5ØØ3Ø HOME : PRINT"ENTER^THE^LINE^NUMBERS^TO^DELETE": VTAB 3
5ØØ4Ø P$ = "DELETE^":X=Ø: GOSUB 5Ø2ØØ:ND=A-1
```

```
50050 HTAB 1: VTAB 1Ø: PRINT"SAVE^ ANY^ EXTRA^ LINES?^ N" CHR$ (8);:
      GET A$: IF A$ < > "Y" THEN 50080
5ØØ6Ø HOME : PRINT"ENTER^ THE^ LINE^ NUMBERS^ TO^ SAVE": VTAB 3
5ØØ7Ø X = 2:P$ = "SAVE^ ": GOSUB 5Ø2ØØ:NI = A - 1
5ØØ8Ø PRINT : PRINT : INPUT "CONTROLLER^ FILENAME=>" ;NM$ : D$ = CHR$
      (13) + CHR$ (4)
5ØØ9Ø PRINT D$ "NOMONCIO": HOME : VTAB 12: PRINT "SAVING^ " NM$
5Ø1ØØ PRINT D$ "OPEN" NM$ D$ "DELETE" NM$ D$ "OPEN" NM$ D$ "WRITE" NM$
5Ø11Ø IF ND = Ø THEN 5Ø13Ø
5Ø12Ø FOR A = 1 TO ND: PRINT "DEL" X(A,Ø) "," X(A,1): NEXT
5Ø13Ø PRINT "DEL1ØØØ,9999": & 1ØØØ,9999
5Ø14Ø IF NI = Ø THEN 5Ø16Ø
5Ø15Ø FOR A = 1 TO NI: PRINT "DEL"X(A,2)","X(A,3): & X(A,2),X(A,3):
      NEXT
5Ø16Ø FOR A = 1 TO LEN (NM$): IF MID$ (NM$,A,1) < > "." THEN NEXT : GOTO
      5Ø18Ø
5Ø17Ø NM$ = LEFT$ (NM$,A - 1)
5Ø18Ø PRINT "?:?" CHR$ (34)NM$"^ CONTROLLER^ ENTERED.";
5Ø19Ø PRINT D$"CLOSE": HOME : VTAB 5: PRINT "CONTROLLER^ SAVED": DEL
      5ØØØØ,59999
5Ø2ØØ FOR A = 1 TO 1Ø: PRINT : PRINT P$;
5Ø21Ø INPUT "";X(A,X),X(A,X + 1): IF X(A,X) = Ø THEN RETURN
5Ø22Ø NEXT : RETURN
```

Dan Rosenburg's & Paul Anderson's APT for...

Miner 2049er

The following is an easier method of entering Miner 2Ø49er on any level than the one that was printed earlier:

When the game asks *HOW MANY PLAYERS?*, type # and a number corresponding to the level (1 - 9 or 0 for level 1Ø) you wish to start on.

Making Liberated Backups That Retain Their Copy-Protection

by Thomas Dragon
(Hardcore COMPUTIST # 7, page 12)

The following is a reasonably efficient procedure for backing-up certain protected disks that are NOT easily copied by either *Locksmith 4.1* or *5.0*.

The 'rub' is that you must **retain** a certain portion of the copy-protection. That is, your copies still can't be duplicated by *Locksmith* or other bit-copy programs. However, it does provide a procedure for backing-up the disks as well as 'liberating' them for study.

Freeing the Disks

1 First, you must have an Apple][with 48K. It is especially convenient if you have an old Integer card living in your Apple, but if you don't, this procedure will still work on a number of protected commercial software programs.

Let us assume that you have an old Integer card. I'll get back to Applesoft later. The switch on the Integer card should be **down** so that Applesoft is the boot-up language.

What's this ? You say that you don't use your Integer card anymore because you have an expansion card in slot 0? Well, clean that old Integer card off because, for our purposes, it will work in ANY slot which allows the switch to stick out of the back of the Apple (slot 0, 2 or 4).

2 Now, pull out the old *DOS 3.3 System Master* and boot up your Apple

```
PR#6
```

Place a blank disk into the drive and INITialize it.

```
INIT HELLO
```

DELETE the HELLO program after you have INITialized the disk.

```
DELETE HELLO
```

This slave disk will serve a very special purpose. Later, we will want to boot the system with it. Unlike a *System Master*, when booting with a slave disk, much of the Apple memory ($4000—$9600) is left undisturbed.

3a Using the Integer Card

Boot the protected disk and watch the screen very carefully during the boot process. If you see the Applesoft prompt for even a fraction of a second the chances are very good that this procedure will liberate the disk.

Flip the Integer switch **up** after the boot process is complete.

Press RESET and you are instantly into the Monitor.

Note that you will always jump into the Monitor no matter where the Integer card is located. This is another well-kept Apple secret.

3b No Integer Card?

If you don't have an Integer card, make like Woody Woodpecker on the RESET key until the disk drive stops.

Then type:

```
CALL -151
```

and you should be in the Monitor. If the drive doesn't stop after 10 or 20 rapid RESETs, then this procedure WON'T work on your Apple. *You might want to see if there is some old-timer who might be willing to sell his useless old Integer card for $15 or $25. There are a number of old Integer cards around that appear not to work. Surprisingly, many of these old cards will still work in terms of forcing a jump to the Monitor. If the old timer has one that doesn't work, just ask him if you can have it to 'study'.*

4 Now that you are in the Monitor, type:

```
A56EG
```

This will activate the CATALOG function if the protected disk has a reasonably intact DOS. You should then see a CATALOG of the disk contents.

If you only get a 'honking', then this procedure won't work. Don't despair. You will very seldom get a honking if you see the Applesoft prompt when you boot.

5 Next, we will move part of the protected RWTS into an area of the Apple memory that will remain intact during the boot process.

In order to accomplish this, we will use the Monitor memory move. Enter:

```
4800<B800.BFFFM
```

This will move the special DOS routines to $4800.

6 Remove the protected disk from the drive and replace it with the slave disk that you previously initialized.

Flip the Integer card switch **down** if you are using an Integer card and boot the slave disk by typing:

```
6 CTRL P
```

(assuming slot 6 is the boot slot). Don't worry about the *FILE NOT FOUND* error. At this point, I am always extra cautious because I don't want to accidentally lose the protected DOS information.. Take out any disk with a bit of space on it and enter the following command from the keyboard

```
BSAVE PROTECTED DOS,A$4800,L$800
```

7 Place the *System Master* into the drive and enter:

```
BLOAD FID
```

Everyone reading this little note must know about *FID*. If you don't, let me just say that it is a DOS-less copy program. It does not carry its own DOS like most other variations of Apple copy routines, but 'sucks' up the DOS that is living in memory when it is activated. However, we have not activated *FID*. We have only loaded it into memory.

8 It is now time to move the protected DOS back into the regular *DOS 3.3* memory area. Enter:

```
CALL -151
```

to get into the Monitor. Then enter:

```
B800<4800.4FFFM
```

Now you have modified *DOS 3.3*. The modified DOS is neither regular *DOS 3.3* nor the true protected DOS. It is a hybrid of regular *DOS 3.3* that employs the protection scheme that is on the protected disk.

9a Return to Applesoft by pressing:

```
CTRL C
```

Now INITialize a slave disk by typing:

```
INIT HELLO
```

If all has gone well, the slave disk will now become a slave disk that retains the copy-protection of the original protected disk. After the drive stops, be sure to:

```
DELETE HELLO
```

9b In case of Error

If you have made an error in moving DOS, just re-boot the slave disk and then go back to Step 7. If you have somehow managed to destroy the memory image, re-boot the system and reLOAD your new DOS:

```
PR#6
BLOAD PROTECTED DOS
```

as saved in Step 6 and then go to Step 7.

1Ø The final step is to activate *FID*. You have the protected disk and you have a blank disk with the same DOS. Either enter:

```
CALL 2Ø51
```

or type:

```
CALL -151
8Ø3G
```

Either CALL will essentially activate *FID* and allow it to use the protected DOS.

11 From this point, you should be able to transfer the files from the protected disk to your special slave with the use of *FID*.

The programs are now essentially liberated from the point of view that they can be LISTed, modified and SAVEd. It may be that the protected HELLO program sets the Reset vector when it is run. If you wish to study the programs, simply load and look at them. To be sure, you should place a write-protect notch on your copied disk if the original copy was write-protected. The disks are still protected in the sense that any copy procedures that failed with the original disk will also fail with the liberated disk.

This procedure works well with the Microzine disks, the Videx pre-boot disks, Micro Power and Light disks, *HIRES Secrets* from Avant Garde, and an incredible collection of other disks that have either binary or Applesoft programs on them. Disks on which the procedure doesn't work include the Human Systems Dynamics (HSD) disks and *E-Z Draw*. It doesn't work for *E-Z Draw* because its disk uses a 32K DOS and this discussion deals with a 48K DOS. It doesn't work for the HSD disks because they have an incredibly complex protection system.

Examining Copy-protected Applesoft BASIC Programs

(Hardcore COMPUTIST # 1Ø, page 24)
By Clay Harrell

Many protected programs are written in Applesoft. Of course, most publishers are sly enough to protect against breaking out of their program with CTRL C or RESET. Also, most protect against re-entering BASIC from the monitor by changing the typical BASIC re-entry point (at $3DØ) so that it points to disaster. Many programs also set the autorun flag at $D6 so if you manage to break out of the program into BASIC, anything you type will RUN their BASIC program. I will describe how to beat all these protection schemes, assuming you have an old style *F8 Monitor ROM* or some means to enter the Apple's Monitor at will.

Is It BASIC?

First, we must determine if the protected program is written in Applesoft. If after you boot the program a BASIC prompt appears, this is a good indication that at least parts of the program are written in BASIC. Furthermore, if the program prints a lot of text on the screen, or requires a good deal of user input it is likely that the program is written in BASIC. The reason for this is that PRINTing text on the screen and INPUTing data from the keyboard is easily accomplished from BASIC using PRINT and INPUT statements. Doing this from assembly language however, requires a great deal more work. Also, we should realize why a programmer uses Assembly language. The only real advantage to Assembly language is speed. If speed is not critical, most non-sadistic programmers will use BASIC.

With this in mind, take note of how the program runs and displays information on the screen. If it runs at about the same speed as the BASIC programs you have written, it is probably written in BASIC. Remember, Assembly language is considerably faster than BASIC in every respect.

Finally, read the package the program came in. The package may say what language it was written in. If it doesn't, a dead give away is in the hardware requirements. If the program requires Applesoft in ROM, then at least part of the program is undoubtedly written in Applesoft.

Viewing the Code

Now that you have figured out that your protected program is written in BASIC, it is time to LIST their code. The first step is to RESET into the monitor after the program has started running.

```
RESET
```

Now you can try to enter the immediate BASIC mode by typing:

```
3DØG
```

The code at $3DØ normally jumps to the BASIC warm start routine but, if the protection scheme is worth anything, this will not work.

Try Again

Assuming that didn't work, reload the program and RESET into the monitor again.

```
RESET
```

The next thing is to try typing:

```
9D84G
```

or typing:

```
9DBFG
```

These are the DOS 'cold' and 'warm' start routines, respectively. If you are lucky enough to get a BASIC prompt, you have done well. Most of the time, however, you will not.

If in either case you succeed in getting a BASIC prompt, try LISTing the program or CATALOGing the disk.

Try, Try Again

If anything you type starts the program running again, the protection has changed the RUN flag at $D6. So RESET into the monitor again.

```
RESET
```

The RUN flag is a zero page location (at $D6) which will automatically RUN the BASIC program in memory if $D6 contains a value of $80 or greater (128 or greater in decimal). This is easy to defeat after you have RESET into the Monitor by typing:

```
RESET
D6:00
```

This resets the RUN flag to normal. Now if 3D0G, 9D84G or 9DBFG previously rewarded you with a BASIC prompt, this will solve the problem of the program re-running when you type a command.

For debugging efforts, the RUN flag can be changed from within a BASIC program by issuing the code:

```
10 POKE 214,255
```

or by POKEing location 214 with anything greater than 127. From Assembly language, the code would most likely look like this:

```
800-  A9 FF      LDA #$FF
802-  85 D6      STA $D6
```

or by loading a register with $80 or greater and storing it at $D6.

Try, Try, and Try Again?

Now if 3D0G, 9D84G or 9DBFG did **not** produce a BASIC prompt, then the DOS in use is more elaborate. So re-load the program and RESET into the monitor after it is running.

Saving the Program

Now come the final steps in trying to examine a BASIC program. If you are using a ROM card in slot zero with an old style *F8 Monitor ROM* to RESET into the Monitor, turn on the mother board ROMs and turn off the ROM card *Integer ROM*s by typing:

```
C081
```

Now reset the RUN flag to normal, just to be sure. Type:

```
D6:00
```

Finally, enter Applesoft the sure fire way by pressing:

```
CTRL C
```

You should see an Applesoft prompt. Now type:

```
LIST
```

and your Applesoft program should now appear.

Strategic Simulation's RDOS

Applying this to a real world example, try this method with one of Strategic Simulation's (SSI) releases. SSI uses a highly modified DOS called *RDOS* for their protection. SSI uses all the tricks mentioned to prevent you from LISTing their programs but, using the above procedure, you can LIST their BASIC programs.

In addition, the DOS used by SSI (RDOS) uses the ampersand (&) in all of its DOS commands. So if you see any ampersands from within their BASIC program, you know it is a DOS command. For example, to CATALOG a SSI disk, after you follow the above procedure and you are in BASIC, type:

```
&CAT
```

This will display SSI's catalog. Very different, eh!

Well, back at the ranch. If you want to SAVE your a protected Applesoft program to a normal DOS disk, follow these steps:

1 RESET into the Monitor after the program is running.

2 If you are using a ROM card in Slot 0, disable its ROMs by typing:

```
C081
```

3 Next turn off the Auto-run Flag and move Page $08 to Page $95 where it will be safe

```
D6:00
9500<800.8FFM
```

4 Check where the Applesoft program ends by typing:

```
AF.B0
```

5 Write down the two bytes that are displayed.

6 Boot a 48K normal *DOS 3.3* slave disk with no HELLO program.

```
PR#6
```

7 Enter the Monitor by typing:

```
CALL -151
```

8 Restore the Applesoft program by typing:

```
800<9500.95FFM
```

BCD: *enter the two bytes you recorded in step 4, separated by spaces.*

9 Enter BASIC and save the program by typing:

```
3D0G
```

SAVE *program name*

What you have done is to move $800—$8FF out of the way so you can boot a slave disk. After normal DOS is up, you restore $800—$8FF from $9500—$95FF, and then restore the end of Applesoft program pointers so DOS knows how big your BASIC program is. Finally, you just SAVEd it to your disk!

Of course there are other more automated ways of getting programs to a normal *DOS 3.3* disk (such as *DEMUFFIN PLUS*), but this is a 'quick and dirty' method that will often work. Keep in mind that the program may not run from normal DOS because of secondary protection within the BASIC program itself. Any curious CALLs, POKEs or PEEKs to memory above 40192 (this is memory where DOS resides) or below 256 (Zero Page memory) should be examined closely.

CATALOGing

Another thing to keep in mind is that the protected disk may have more than one file on it. If the protected DOS's commands have been modified you may not be able to CATALOG the disk directly to see how many files are on it. If this is the case, you can try to execute the CATALOG command handler at $A56E. To do this, boot up with the protected disk and RESET into the Monitor.

```
RESET
```

Then type:

```
A56EG
```

With any luck at all, the disk's directory should be displayed. If there is more than one file you will probably want to transfer all the files to normal DOS with the use of a program such as *DEMUFFIN PLUS*.

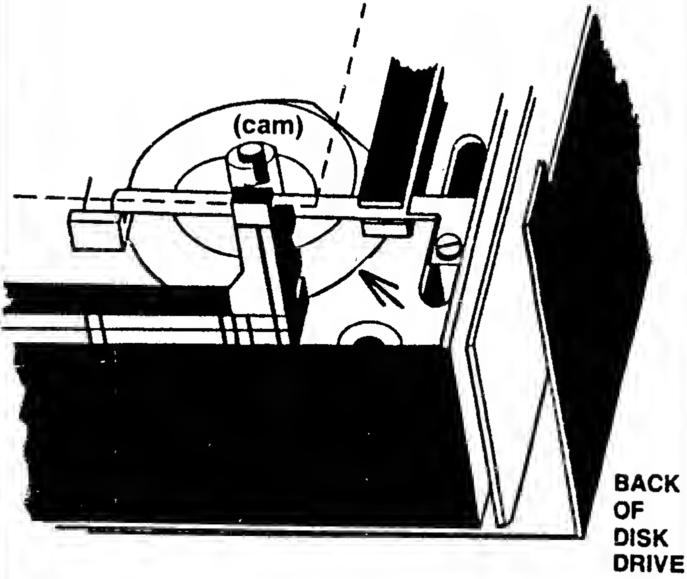
If you experiment with the techinques I have described in this article I think you will be surprised at the number of programs whose copy protection can be fairly easily removed.

ERRATA FOR...

The Book Of Softkeys Volume I

This figure (omitted in The Book Of Softkeys Volume I) is for paragraph 3 of **'Getting On The Right Track'** *by Robert Linden* (Book Of Softkeys Volume I, page 99).

Figure 1



The editorial staff of SoftKey Publishing would like to extend our apologies. We sincerely regret any inconvenience this omission may have caused you.

YES!

We still have

Volume I of

The Book Of Softkeys

YES, I want Volume I of The Book Of Softkeys. I have enclosed $12.95 per book. For shipping/handling per book, add $2 for domestic orders and $5 for foreign orders. U.S. funds drawn on U.S. banks. Washington State orders add 7.8% sales tax. Send your orders to:

SoftKey Publishing PO Box 110937-BK Tacoma, WA 98411

Name ____________________

Address ____________________

City ____________________ State __________ Zip __________

Country ____________________ Phone ____________________

__________-__________-__________-__________ Exp. __________

Signature ____________________ BSK2

The Book Of Softkeys Volume I

contain softkeys for:

Akalabeth, Ampermagic, Apple Galaxian, Aztec, Bag of Tricks, Bill Budge's Trilogy, Buzzard Bait, Cannonball Blitz, Casino, Data Reporter, Deadline, Disk Organizer II, Egbert II Communications Disk, Hard Hat Mack, Home Accountant, Homeword, Lancaster, Magic Window II, Multi-disk Catalog, Multiplan, Pest Patrol, Prisoner II, Sammy Lightfoot, Screen Writer II, Sneakers, Spy's Demise, Starcross, Suspended, Ultima II, Visifile, Visiplot, Visitrend, Witness, Wizardry, Zork I, Zork II, Zork III...

plus how-to articles

and program listings of need-to-have programs used to make deprotected backups.